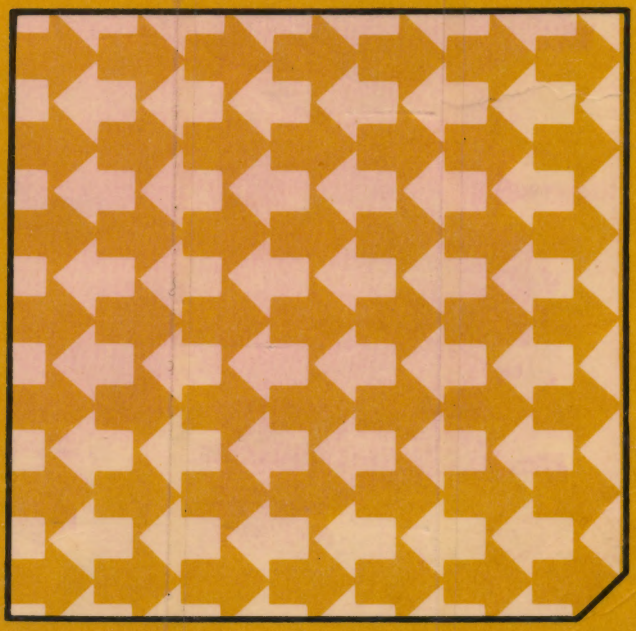
МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Ч. Хьюз, Ч. Пфлигер, Л. Роуз

МЕТОДЫ ПРОГРАММИРОВАНИЯ: КУРС НА ОСНОВЕ ФОРТРАНА



ИЗДАТЕЛЬСТВО МИР

ИЗДАТЕЛЬСТВО МИР

ADVANCED PROGRAMMING TECHNIQUES

A SECOND COURSE IN PROGRAMMING USING FORTRAN

Charles E. Hughes
University of Tennessee

Charles P. Pfleeger
University of Tennessee

Lawrence L. Rose
Ohio State University

JOHN WILEY & SONS
NEW YORK SANTA BARBARA CHICHESTER
BRISBANE TORONTO
1978

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Ч. Хьюз, Ч. Пфлигер, Л. Роуз

МЕТОДЫ ПРОГРАММИРОВАНИЯ: КУРС НА ОСНОВЕ ФОРТРАНА

Перевод с английского
Ю. В. Ступина

под редакцией
Ю. М. Баяковского

ИЗДАТЕЛЬСТВО
«МИР»
МОСКВА 1981

УДК 681.142.2
518
Х 98

Хьюз Ч., Пфлигер Ч., Роуз Л.

Х 98 Методы программирования: курс на основе Фортрана. Пер. с англ. Ю. В. Ступина / под ред. и с предисл. Ю. М. Баяковского.—М.: Мир, 1981.
336 с., с ил.

Известные американские авторы ставят своей целью помочь начинающим программистам в совершенствовании техники программирования на языке Фортран IV. Рассмотрены структура языка, методы повышения эффективности программ, ввод/вывод на магнитных лентах и дисках, подпрограммы, неарифметические операции, структуры данных. Приводятся краткие сведения о языке Фортран 77 и управлении заданиями в операционных системах.
Для всех, кто изучает язык Фортран или программирует на нем.

Х $\frac{20204\text{-}035}{041\ (01)\text{-}81}$ 35-81, ч. 1. 2405000000 681.142.2

Редакция литературы по математическим наукам

© 1978 by John Wiley & Sons, Inc.
All Rights Reserved
Authorized translation from English language edition published by John Wiley & Sons, Inc.

© Перевод на русский язык, «Мир», 1981

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

На глазах одного поколения людей уже сменились три поколения вычислительных машин. Одновременно с увеличением мощности машин росла и сложность решаемых задач. Программы, которые 20 лет назад относили к разряду серьезных (и даже научных) достижений, теперь сравнимы с заурядной курсовой работой. Программирование перестало быть делом одиночек, в реализации современных проектов участвуют порой сотни специалистов. Взять хотя бы такие факты: на создание системы с разделением времени (TSS/360) было затрачено 2000 человеко-лет, за 5 лет разработки система претерпела 8 кардинальных ревизий, причем в последней ее версии почти ничего не осталось от первой.

В сложившихся условиях исключительно важное значение приобретают инженерные и организационные аспекты программирования. Однако в этом отношении выпускники высших учебных заведений недостаточно подготовлены. Они испытывают трудности в совместной работе с другими программистами, занятыми в проекте, не владеют техникой систематического тестирования написанной программы, плохо и несвоевременно ее документируют, не умеют писать структурные программы на неструктурном языке, не способны планировать работу и укладываться в график проекта.

В США одним из важных этапов на пути решения этой проблемы стала разработка единых учебных планов (первый их вариант был опубликован в 1968 г., второй — десять лет спустя). Среди 8 основных курсов лекций, предусмотренных этими учебными планами, два обязательных вводных курса. В первом из них рассматриваются основные понятия и изучается язык программирования. Второй курс посвящен структурам данных, методам конструирования, отладки и тестирования программ. Этот второй курс и составляет предмет настоящей книги.

Для записи алгоритмов авторы использовали язык Фортран. Книга писалась в ту пору, когда еще не был утвержден новый стандарт — Фортран 77, и поэтому большинство примеров дается на Фортране IV (стандарт 1966 г.), лишь иногда применяются средства расширенного диалекта — WATFIV. В приложении

приводится перечень изменений, внесенных в язык Фортран 77, но этот перечень не полон и не совсем точен. Отсутствует, например, описание новой системы ввода/вывода, не упоминается инструкция PARAMETER и т. д.

Идеи структурного программирования невозможно продемонстрировать на коротких примерах, поэтому в книге имеется много довольно пространных программ. Они являются неотъемлемой частью текста (как формулы или выкладки в математическом тексте), и при чтении книги их следует внимательно изучать.

Принято считать, что язык Фортран предназначен для решения задач научного и инженерного характера. Однако в данном учебнике большинство примеров относится к управленческо-экономической области, т. е. к сфере «нечисленных» приложений. Читатель, возможно, неожиданно для себя обнаружит, что и в этом случае Фортран оказывается вполне приемлемым и даже удобным инструментом.

В заключение отметим, что в работе над переводом книги принимала участие Н. И. Вьюкова (гл. 5).

Ю. Баяковский

ПРЕДИСЛОВИЕ

В книге представлен материал, который принято относить к курсу программирования второй ступени. В ней учтена бо́льшая часть рекомендаций, содержащихся в пересмотренной учебной программе ACM 68[1]).

Авторы стремились собрать воедино материал, который позволяет начинающему программисту достичь профессионального уровня, а профессиональному программисту повысить свою квалификацию. Поэтому вначале обсуждаются понятия и принципы, а затем они иллюстрируются развернутыми примерами. Примеры отражают разнообразие стилей в конструировании и документировании программ.

Для реализации алгоритмов используется язык Фортран. В книге описаны характерные особенности диалекта WATFIV, а также тех диалектов американского стандартного Фортрана IV (1966 г.), которые реализованы на машинах IBM 360-370 и DEC-10. Там, где это необходимо, отмечаются различия между Фортраном IV и новым стандартом — Фортраном 77.

Книга может быть полезна как студентам, специализирующимся в области информатики (программирования), так и различным пользователям ЭВМ. От читателей не требуется каких-либо специальных знаний, кроме тех навыков программирования на любом процедурно-ориентированном языке (не обязательно на Фортране), которые они приобрели, прослушав первый вводный курс.

Книга состоит из девяти глав и двух приложений. Главы 0 и 1 заменяют введение. В гл. 0 рассматриваются основы языка Фортран; этот материал используется в последующих главах. Материал изложен сжато; те, кто знаком с языком Фортран, могут использовать эту главу как справочник, тот же, кто знает какой-либо другой процедурно-ориентированный язык, найдет в ней необходимые начальные сведения. В гл. 1 изучаются такие вопросы, как выбор алгоритма, разработка программы, стиль программирования, стандарты на документацию, методы отладки и эксплуатации программ.

---

[1]) Ассоциация вычислительной техники (Association for Computing Machinery). — *Прим. ред.*

В гл. 2—4 описаны более мощные средства языка Фортран. В гл. 2 речь идет о процедурах, при этом особое внимание уделено методам разделения данных. Глава 3 посвящена обработке текстовой информации. В гл. 4 описаны организация и использование магнитных лент и дисков, а также инструкции Фортрана для работы с указанными устройствами. Кроме того, мы знакомим читателей с некоторыми возможностями командных языков в операционных системах IBM 360-370 и DEC-10.

В примерах, содержащихся в гл. 0—4, используются структуры данных, которые естественным образом представляются простыми переменными и массивами. В гл. 5 приведены методы представления и обработки более сложных структур данных (связанных списков, стеков, очередей и деревьев).

В гл. 6—8 рассматривается аппаратная и программная среда, в которой выполняется программа пользователя. В гл. 6 описано внутреннее представление данных в машине. Элементарные сведения об организации машины содержатся в гл. 7. Наконец, в гл. 8 обсуждаются методы снижения стоимости разработки, эксплуатации и выполнения программ. В частности, мы знакомим читателей с такими понятиями, как объектная колода, перемещение программы, загрузка и оверлейные структуры.

В приложении 1 перечислены встроенные функции большинства диалектов Фортрана. В приложении 2 отмечены особенности языка Фортран 77 по сравнению с языком Фортран IV, который изложен в различных главах этой книги, а по ходу изложения сделаны специальные пометки там, где в языках Фортран IV и Фортран 77 имеют место различия. Они прокомментированы в приложении.

Приведенные в книге примеры намеренно отражают разнообразие стилей программирования. Использовано несколько различных методов разработки алгоритма, конструирования программы и ее документирования. Мы не считаем, что существует какой-то один стиль программирования, который во всех отношениях лучше, чем другие. Любой стиль имеет свои достоинства и свои сферы приложения. Хотелось бы, чтобы каждый программист познакомился с различными приемами программирования, выделил наиболее интересные из них и разработал удобный для себя стиль.

Мы хотим выразить признательность за многие полезные советы, которые высказали Эл Девис, Билл Хайнес, Роберт Мак-Гаффи и рецензировавшие книгу Дж. Мэк Адамс, Фрэнк Проссер и Бен Шнейдерман. Мы хотим поблагодарить также многих студентов, которые проработали различные версии этого учебника и чья критика улучшила настоящее издание. Наконец, мы хотим выразить наше признание и благодарность Джойс Марляр, которая мастерски напечатала бóльшую часть рукописи.

Конечно, мы берем на себя ответственность за все оставшиеся незамеченными ошибки и упущения.

Представленные в книге программы проверены на ЭВМ IBM 360 модели 65 в Вычислительном центре Университета штата Теннесси. Некоторые из них дополнительно проверены на ЭВМ DEC-10 с процессором KL Университета штата Теннесси и на ЭВМ DEC-10 с процессором KA, работающей на факультете информатики Университета штата Огайо.

Чарльз Е. Хьюз
Чарльз П. Пфлигер
Лоуренс Л. Роуз

Глава 0

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

0.1. Обзор

Эта книга поможет вам стать по-настоящему грамотным программистом. В ней на примерах разобраны те методы, которые применяют опытные программисты при решении своих задач. Изложение основано на языке Фортран, но многое из того, о чем вы узнаете в этой книге, применимо и к другим алгоритмическим языкам. Вы познакомитесь со многими аспектами вычислений, не зависящими от языка программирования.

По мере накопления опыта программисты приобретают свой «стиль», т. е. у них вырабатываются определенные привычки, проявляющиеся в структуре программы, ее документировании и т. д. Гл. 1 посвящена стилю программирования, но и по ходу всей книги мы будем отмечать особенности, присущие хорошему стилю. В гл. 2—4 рассмотрены такие вопросы, как использование подпрограмм, обработка текстов, операции с устройствами ввода/вывода на магнитных лентах и дисках. В последующих главах вы познакомитесь со средой, в которой выполняются программы; в них также рассмотрены вопросы внутреннего представления данных, структуры ЭВМ и процесса выполнения программы.

Возможно, вы никогда не изучали Фортран или программировали на нем много лет назад. Эта книга была задумана так, чтобы ее поняли все, кто имеет опыт программирования на каком-либо процедурно-ориентированном языке. Если вы знакомы с языками ПЛ/1, Алгол, Паскаль или Кобол, вам не составит никакого труда изучить Фортран. Вводная глава содержит материал, который вам потребуется в дальнейшем. Даже в том случае, если вы уже знаете Фортран, мы советуем прочесть эту главу, хотя бы для того, чтобы освежить материал в памяти и убедиться в том, что вы действительно знаете основы Фортрана. Описание каждой инструкции дано в виде завершенного подраздела и выделено на фоне остального текста так, чтобы такие подразделы можно было быстро найти.

Язык Фортран описан в соответствии со стандартом Института Американских национальных стандартов (сообщение X3.9—1966). Позже был предложен новый стандарт языка, расширяющий его возможности; новая версия названа Фортран 77. Там,

где есть различия между этими двумя версиями языка, абзацы начинаются и заканчиваются черным кружочком; в приложении 2 вы найдете описание этих отличий.

Некоторые особенности Фортрана зависят от типа ЭВМ и компилятора. В ряде компиляторов принята лишь часть стандарта, в других — целиком весь стандарт Фортрана; есть и такие компиляторы, которые допускают расширения стандарта. Будем называть каждую комбинацию машины и компилятора *диалектом*. Многие из приведенных примеров, зависящих от реализации, основаны на двух широко распространенных диалектах: для машин IBM 360-370 с компилятором WATFIV и машин DEC-10 с компилятором Фортран-10.

0.2. Основные понятия и определения

0.2.1. Запись инструкций Фортрана

Программа на Фортране записывается в виде последовательности инструкций. Инструкции имеют фиксированный формат и вводятся в машину как записи; в каждой записи не более одной инструкции. Формат инструкций показан на перфокарте, рис. 0.1. Хотя в действительности ввод в ЭВМ происходит с

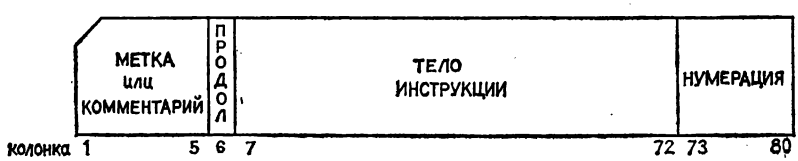


Рис.0.1. Формат инструкций Фортрана.

различных устройств, например с клавиатуры терминала, с перфоленты и др., мы будем говорить о входных записях как о перфокартах.

Метка (или *номер инструкции*) Фортрана — это число, расположенное где угодно в пределах колонок с первой по пятую. Метками снабжают те инструкции, на которые есть ссылки в программе. Другие инструкции программы записывают без меток.

Тело инструкции перфорируют в колонках с 7 по 72. Для удобства чтения программы в тело инструкции можно вставлять пробелы. Компилятор их игнорирует (кроме пробелов внутри литерных данных, о чем будет сказано ниже). Тело инструкции может начинаться с любой, не обязательно с 7-й колонки.

Если тело инструкции настолько длинно, что не умещается на одной перфокарте, инструкция может быть продолжена на следующей перфокарте. В этом случае пробивают любой, отлич-

ный от пробела или нуля символ в *колонке продолжения* (колонка 6) следующей перфокарты и продолжают инструкцию в колонках с 7 по 72 этой карты. Если необходимо, инструкция может быть аналогично продолжена на следующие перфокарты. Одна инструкция может располагаться максимально на 20 перфокартах (19 карт продолжения).

Колонки с 73 по 80 используют для *нумерации последовательности* перфокарт, но такая нумерация необязательна. Для некоторых входных носителей, отличных от перфокарт, этого поля или совсем нет, или оно заполняется пользователем по его усмотрению. Для больших программ в этом поле удобно пробивать номера, чтобы иметь возможность отсортировать входную колоду, если перфокарты будут перепутаны.

• Колонку 1 используют для указания *комментария*. Любая карта с буквой С в колонке 1 рассматривается как комментарий и игнорируется компилятором. Комментарии позволяют идентифицировать программу, наглядно разделять ее на логические сегменты, объясняют назначение этих сегментов •

0.2.2. Типы данных

Элементы данных Фортрана хранятся в словах фиксированной длины. Слово памяти состоит из определенного числа двоичных разрядов (0 или 1), называемых *битами*. Слово в машинах IBM 360-370 состоит из 32 битов; слово машины DEC-10 имеет 36 битов. Слово может содержать один или несколько элементов данных в зависимости от их типа; в некоторых ЭВМ для представления одного элемента данных используется несколько слов. Первые четыре типа данных в Фортране, с которыми мы познакомимся, это целые, вещественные, логические и литерные.

Данные целого типа — это такие числа, у которых нет дробной части. Под *целой константой* понимают последовательность из одной или более десятичных цифр без десятичной точки, которой может предшествовать знак + или —. Примеры целых констант: +2, —9695, 800000.

Вещественные данные — это числа, которые могут содержать дробную часть. *Вещественные константы* представляются в двух формах: в виде последовательности десятичных цифр с десятичной точкой; в виде последовательности десятичных цифр (с десятичной точкой или без точки), за которой следует буква Е и целая константа. В обоих видах записи вещественной константе может предшествовать знак + или —. Примеры вещественных констант: 1., +3.1416, —.483695 и +0.428E2, —369E6, 1.0E—4. В последнем случае вещественные константы представлены в так называемой «научной нотации». Здесь величина, следующая за

E, показывает, в какую степень надо возвести число 10 перед умножением на первую часть числа. Так, последние три примера представляют числа 42.8, —369000000., .0001. Форма представления с порядком обычно применяется для записи очень больших или очень малых чисел.

Логические данные могут принимать значения «истина» или «ложь»; константы, соответствующие этим значениям, записывают в виде .TRUE. и .FALSE.

● Литерные данные представляют собой последовательность букв, цифр, некоторых знаков и пробелов. *Литерная константа* — цепочка литер имеет вид *n*H*const*, где *n* — целая константа без знака, указывающая длину (число литер) цепочки литер, а *const* — цепочка из *n* литер после буквы H. Примеры такого типа констант: 5HCPP01 и 11HCAN'T SPELL. Во многих диалектах Фортрана цепочка литер литерной константы заключается в одиночные кавычки (апострофы). Внутри такой константы знак «апостроф» представляется двумя последовательными кавычками. Приведем два примера: 'IBM OS/360' и 'ISN''T BAD'. Действия с литерными данными описаны в гл. 3. Мы будем использовать литерные константы только для записи заголовков при выводе информации ●

Тип ЭВМ	Диапазон целых чисел (в разрядах)	Диапазон вещественных чисел	Точность вещественных чисел (в разрядах)	Число литер в слове
Burroughs 6700	12	10^{-46}—10^{+69}	11	8
CDC 6000-7000	18	10^{-293}—10^{+322}	15	10
DEC-10	10	10^{-38}—10^{+38}	8	5
IBM 360-370	9	10^{-78}—10^{+75}	7	4
UNIVAC 1108-1110	10	10^{-38}—10^{+38}	8	6
XEROX SIGMA 5-7	9	10^{-78}—10^{+75}	7	4

Рис.0.2. Диапазон и точность чисел, представление литер.

Диапазон значений, которые могут принимать данные указанных типов, зависит от размера слова конкретной вычислительной машины. На рис. 0.2 показан диапазон значений, который могут принимать числа в некоторых распространенных машинах. Диапазон целых величин *n* разрядов указывает, что допустимые в машине целые величины могут состоять не более, чем из *n* цифр. Значение показателя степени характеризует диапазон вещественных величин. Точность представления чисел в этих машинах определяется указанным на рис. 0.2 количеством десятичных разрядов. Число литер, которое может быть записано в машинном слове, зависит от способа хранения литерных данных в оперативной памяти; об этом мы подробно поговорим в гл. 3.

Переменные в Фортране могут быть целыми, вещественными или логическими (Фортран 77, WATFIV и многие другие диалекты допускают также литерные переменные). Тип переменной не изменяется во время выполнения программы.

Именем переменной в Фортране может быть любая последовательность, содержащая от одной до шести букв или цифр, причем первой литерой в этой последовательности должна быть буква. Так, TOTAL, I, SUMOFX и COL72 — это правильно записанные имена переменных. Имя переменной в Фортране может идентифицировать простую переменную, элемент массива или массив. *Простая переменная* используется для представления отдельных величин и связана с одним словом памяти.

Каждый *элемент массива* также представляет какую-то одну величину. Элементы массива обозначаются в Фортране с помощью индексов, располагаемых в скобках непосредственно после имени массива. Максимально массив переменных может быть трехмерным, но в некоторых компиляторах допускаются массивы и бóльших размерностей. Если, например, ROWSUM — одномерный массив (называемый также вектором) из 10 элементов, то шестой элемент обозначается как ROWSUM (6). Индекс должен быть целым числом, меньше или равным объявленному максимальному количеству элементов массива. В Фортране каждый массив начинается с элемента с индексом «1»; индексы не могут быть отрицательными или принимать нулевое значение. Так, если I — простая целая переменная, то ROWSUM(I), ROWSUM (I+2), ROWSUM(3*I+15) и т. д.— это правильное обращение к вектору ROWSUM; разумеется, при вычислении индекса результат должен быть не меньше 1 и не больше числа элементов массива.

• Допускаемая сложность выражений для вычисления индекса зависит от компилятора. В ряде компиляторов для указания значения индекса нельзя пользоваться выражениями, более сложными, чем целая константа без знака, умноженная на простую целую переменную плюс или минус целая константа без знака. В ряде других компиляторов для вычисления индекса допускаются любые целочисленные выражения. Необходимо знать ограничения, присущие компилятору, с которым вы работаете •

Имя массива ассоциируется с группой последовательно расположенных слов (по одному на каждый элемент массива). Элементы массива хранятся в памяти *по столбцам*. Это означает, что в последовательных словах памяти вначале запоминаются элементы первого столбца, затем второго и т. д. На рис. 0.3 показано, как хранятся простые переменные I, J, KOUNT, MAX, четыре элемента вектора TOP и массив SUM размером 3×2.

Когда в программе происходит обращение к элементу массива, вычисляется индекс и по его значению находится слово па-

мяти, связанное с данным элементом массива. Эта связь устанавливается с помощью формулы доступа и таблицы символов. *Таблица символов* — это таблица имен и связанных с ними адресов. Адрес определяет первое слово области, в которой хранится переменная или массив с данным именем.

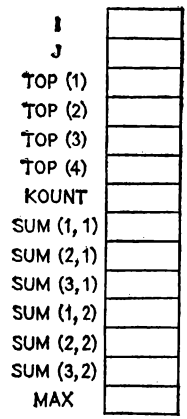


Рис. 0.3. Расположение данных в памяти машины.

Формула доступа — это формула, позволяющая установить, какое слово памяти связано с данным элементом массива. Напомним, что в Фортране каждый массив хранится в виде блока последовательных слов, и элементы массива расположены в памяти машины по столбцам. Так, элемент A(K) и элемент A(1) разделяют (K—1) слов. В двумерном массиве для того, чтобы определить положение конкретного элемента массива, надо установить, сколько столбцов предшествуют столбцу, содержащему данный элемент, и затем подсчитать, сколько элементов в этом столбце предшествуют этому элементу. Если X — это массив размером M×N, то адрес элемента X(I, J), относительно начала массива X определяется как

$$\text{Адр}\ [X(I,\ J)] = (J-1)*M + (I-1)$$

Таким образом, элемент X(I, J) находится в ячейке памяти, которой предшествует число слов, занятых столбцами, предшествующими столбцу J (J—1 столбцов длиной M) плюс число слов, занятых элементами столбца J, предшествующими строке I (I—1 элементов).

При расположении в памяти многомерного массива по описанному выше принципу (первый индекс меняется быстрее второго) вначале располагаются элементы первого измерения массива (по столбцам), затем второго и т. д. Например, элементы массива Y размером 2×3×2 будут расположены в памяти в следующем порядке: Y(1, 1, 1), Y(2, 1, 1), Y(1, 2, 1), Y(2, 2, 1), Y(1, 3, 1), Y(2, 3, 1), Y(1, 1, 2), Y(2, 1, 2), Y(1, 2, 2), Y(2, 2, 2), Y(1, 3, 2), Y(2, 3, 2). При распределении памяти компилятор Фортрана использует декларированный размер массива, для того чтобы отвести ему соответствующее количество последовательных слов памяти. Во время выполнения программы как размерность массива, так и максимальное число элементов массива остаются неизменными. Это связано с тем, что в реализациях Фортрана применяется стратегия *статического распределения* памяти, при которой области памяти под массивы или простые переменные отводятся до выполнения программы и размеры этих областей не изменяются во время выполнения.

0.2.3. Операции

В Фортране приняты три категории операций: арифметические, логические и операции отношения. Арифметические операции используются в вычислениях с целыми или вещественными величинами. Арифметические операции показаны на рис. 0.4.

Знак операции	*Смысл*
**	возведение в степень
+, —	плюс, минус
* /	умножение, деление
+, —	сложение, вычитание

Рис.0.4. Арифметические операции.

Символ (**) означает возведение в степень (например, X**2 означает X в квадрате, или X в степени 2). Символы плюс и минус указывают знак величины (например, +3.75 или—2) или отрицательное значение величины (например, —A). Четыре другие арифметические операции (*/ + —) общеизвестны.

Операции отношения, показанные на рис. 0.5, используются для сравнения двух величин, которые могут быть переменными, константами или выражениями. С помощью этих операций можно проверить следующие условия:

Знак операции	*Смысл*
.LT.	меньше чем
.LE.	меньше или равно
.GT.	больше чем
.GE.	больше или равно
.EQ.	равно
.NE.	не равно

Рис.0.5. Операции отношения.

Каждая операция отношения обозначается четырьмя литерами (включая точки).

• Логические операции приведены на рис. 0.6.

Знак операции	*Смысл*
.NOT.	логическое “не”
.AND.	логическое “и”
.OR.	логическое “или”

Рис.0.6. Логические операции.

Эти операции применяются для построения сложных логических выражений. Логическое «не» отрицает значение логического выражения; две другие операции объединяют два логических выражения в одно составное логическое выражение •

0.2.4. Выражения

В Фортране различают два типа выражений: арифметические и логические. К *арифметическим выражениям* относятся:

1. Целая или вещественная константа.
2. Целая или вещественная переменная.
3. Два арифметических выражения, связанные знаком арифметической операции.
4. Арифметическое выражение, заключенное в скобки.
5. Арифметическое выражение, которому предшествует знак + или —. Заметим, однако, что два знака арифметических операций не должны стоять рядом, поэтому, например, «А умножить на минус 10» должно быть записано так: А*(—10).

Способ записи арифметических выражений близок к тому, который является общепринятым в математике. Скобки применяют для управления последовательностью выполнения операций, а также для выделения индексов в массивах. В арифметическом выражении могут встречаться только вещественные переменные и константы или только целые операнды либо комбинации вещественных и целых операндов. Результат вычисления арифметического выражения с однотипными операндами (все операнды одного типа — только целые или только вещественные) определяется типом операндов в выражении. Однако выражения, содержащие и целые, и вещественные операнды (такие выражения называют *выражениями смешанного типа*), дают в результате вычисления вещественное значение. Каждый раз, когда в выражении участвует вещественная величина, результат вычисления — величина вещественного типа.

Расположение скобок и приоритет операций управляют последовательностью вычислений в выражении. Следующие четыре правила определяют порядок, в соответствии с которым вычисляются арифметические выражения.

1. Величины внутри скобок должны быть вычислены прежде, чем они могут участвовать в операциях совместно с какими-либо другими величинами вне скобок.

2. Операции внутри арифметического выражения выполняются в порядке их приоритета: возведение в степень **, затем одноместные операции + и —, далее умножение * и деление / и в последнюю очередь сложение + и вычитание —. В этом же порядке операции перечислены на рис. 0.4.

3. Вычисление выражений, имеющих операции одного приоритета (кроме возведения в степень), производится слева направо в порядке записи операций.

4. В выражениях, содержащих две последовательные операции возведения в степень (например, A**B**C), должны быть расставлены круглые скобки, определяющие порядок вычислений, поскольку порядок вычислений в такой ситуации не одинаков для различных компиляторов Фортрана.

Примеры: Пусть J=7 (целое), I=5 (целое), X(6)=3 (вещественное).

	выражение	*результат*	*тип*
(а)	I*3.1	15.5	веществ.
(б)	I — X (J — 1)	2.0	веществ.
(в)	I + J/2 * 5	20	целый
(г)	— X(I + 1) * * 2/2	—4.5	веществ.
(д)	(X(6) + 1.0) * 4.0	16.0	веществ.

В примерах (а) и (б) показаны арифметические выражения смешанного типа; результат вычисления этих выражений — вещественные величины. В примере (в) деление и умножение имеют более высокий приоритет, чем сложение, поэтому они выполняются первыми слева направо в соответствии с правилами 2 и 3: 7/2=3, 3*5=15, 5+15=20 — конечный результат. Заметим, что деление двух целых дает в результате целое, а остаток от деления отбрасывается. Мы говорим, что в результате деления целого остаток *усекается:* поэтому 7/2 равно 3. В примере (г) вначале выполняется операция возведения в степень (она имеет наивысший приоритет), за ней следует операция изменения знака и наконец (последним) выполняется деление. Эти вычисления можно записать так: (—(3.**2))/2=—4.5. В примере (д) показано, как применяются скобки для изменения порядка вычислений: здесь сложение выполняется раньше умножения: (3.+1.)*4 или 4.*4.=16.

Теперь рассмотрим логические выражения Фортрана. К *логическим выражениям* относятся:

1. Логическая константа.
2. Логическая переменная.
3. Два арифметических выражения, разделенные символом операции отношения.
4. Логическое выражение, заключенное в скобки.
5. Логическое выражение, перед которым стоит символ логической операции .NOT. .
6. Два логических выражения, разделенных символом логической операции .AND. или .OR.

Логические выражения вычисляются в следующем порядке:

1. Части выражения, заключенные в скобки, вычисляются раньше тех частей, которые находятся вне скобок.

2. Любые арифметические выражения, входящие в состав логических выражений, вычисляются первыми.
3. Операции отношения выполняются после арифметических операций (логические операции выполняются в последнюю очередь).
4. Наконец, логические операции выполняются слева направо в порядке их приоритета: .NOT., .AND. и затем .OR. . (Именно в таком порядке логические операции перечислены на рис. 0.6.)

Примеры: Предположим, что логические переменные ON и A имеют соответственно значения .TRUE. и .FALSE. и целая переменная I имеет значение 5.

	выражение	*результат*
(а)	I. LT.5	.FALSE.
(б)	ON. AND. .NOT.A	.TRUE.
(в)	A .OR. (I .GE. 3) .AND. ON	.TRUE.

В примере (а) I не меньше 5, поэтому результат .FALSE. . В соответствии с правилом 4 операция .NOT. в примере (б) выполняется раньше .AND. Первое вычисленное значение .NOT. A дает .TRUE. , а затем ON и (.NOT. A), связанные логической операцией .AND., дают в результате .TRUE. В примере (в) по правилу 1 вначале вычисляется (I .GE. 3). Результат вычисления этого выражения .TRUE. . (Если даже опустить скобки, в соответствии с правилом 3 этот терм будет вычисляться первым.) Затем по правилу 4 выполняется операция (I .GE. 3) .AND. ON, результат которой .TRUE. . И наконец, A .OR. (.TRUE.) дает .TRUE. .

0.2.5. Условные обозначения

Мы подошли теперь к тому, чтобы начать изучение основных инструкций языка Фортран. Запись инструкций должна точно соответствовать приведенным ниже описаниям вплоть до каждой прописной буквы и знака препинания. Части, *выделенные курсивом*, должны замещаться при написании программы конкретным содержанием. В квадратные скобки заключаются элементы инструкции, которые могут быть опущены. В самих инструкциях квадратные скобки, конечно, отсутствуют. Например, условная запись

[*метка*] IF (*лог-выр*) *инстр*

указывает, что инструкция может содержать метку, обязательно содержит ключевое слово IF, за которым в круглых скобках следует элемент *лог-выр* и далее *инстр*. Обозначения *метка*,

лог-выр и *инстр* будут объяснены позднее при описании инструкций.

Пробелы в инструкции компилятор Фортрана игнорирует, за исключением пробелов внутри литерных констант. Поэтому советуем вам при записи инструкций свободно пользоваться пробелами везде, где они позволяют сделать программу более удобной для чтения.

0.3. Неисполняемые инструкции

Большинство инструкций в программе, написанной на Фортране, относится к типу *исполняемых* инструкций. Это означает, что такие инструкции преобразуются в машинные команды и вызывают определенные действия в ЭВМ во время выполнения программы. Другой класс составляют *неисполняемые* инструкции Фортрана. Они не транслируются в машинные команды, а только обеспечивают необходимую информацию для компилятора во время трансляции. В данном разделе рассмотрены неисполняемые инструкции Фортрана.

0.3.1. Декларации

Рассмотрим первый вид неисполняемых инструкций Фортрана — так называемые инструкции описания, или декларации. Они управляют типом, размерностью, начальным значением и размещением переменных внутри программы. В этой главе будут описаны неисполняемые инструкции DIMENSION, REAL, INTEGER, LOGICAL, IMPLICIT и DATA. Инструкция EQUIVALENCE описана в гл. 1, инструкции COMMON и функции-формулы — в гл. 2. Декларации должны находиться в самом начале программы (или подпрограммы; см. разд. 0.7), так как они определяют способ интерпретации компилятором следующих за ними исполняемых инструкций.

В начале программы необходимо указать максимальное значение каждого индекса массива, используемого в программе. Если *размерность* переменной не указана, переменная классифицируется как простая (безындексная). Размерность массива не может изменяться во время выполнения программы. Размерность массива можно задать с помощью инструкции DIMENSION:

DIMENSION *список*

где *список* — последовательность индексируемых переменных, разделенных запятыми.

Пример:

```
DIMENSION X(10), SUM(2, 3), TOP(5)
```

Действие: Эта декларация резервирует 10 слов для вектора X, 6 слов (2*3) для двумерного массива SUM и 5 слов для вектора TOP.

Примечания:

1. В программе может быть несколько деклараций DIMENSION.
2. Число измерений, разрешенных для массива, зависит от компилятора.
3. Индексы должны быть целыми константами и соответствовать максимальному значению для каждого измерения.
4. Каждое имя массива должно быть единственным в программе, т. е. им можно пользоваться только для ссылок на данный массив.

• Как мы уже видели, переменные могут быть целыми, вещественными или логическими. Тип переменной должен быть указан компилятору Фортрана, чтобы в объектной программе можно было правильно интерпретировать двоичные данные в слове, связанном с переменной. Если программист не объявил тип переменной, компилятор принимает тип в соответствии с первой буквой имени переменной. Следующие три декларации служат в Фортране для явного указания типа переменной •

REAL *спис-перем*
INTEGER *спис-перем*
LOGICAL *спис-перем*

где *спис-перем* — *список переменных* — это имена простых переменных, массивов и функций, разделенных запятыми. (В некоторых компиляторах в декларациях типа переменных разрешается задавать информацию о размерности. В этом случае указанные в *спис-перем* индексированные переменные содержат индексы, обозначающие размерности.)

Пример:

REAL X, KOUNT, ROW (6)

Действие: эта декларация объявляет переменные X, KOUNT и ROW вещественными и ROW линейным массивом из шести элементов.

Примечания:

1. В программе может быть несколько деклараций REAL, INTEGER или LOGICAL.
2. Имя переменной может появиться лишь однажды, в какой-либо одной декларации типа.
3. В качестве индексов могут быть заданы только целые

константы. Они должны представлять максимальное значение индекса в соответствующем измерении.

4. Если размерность массива указана в инструкции описания типа, то массив можно уже не упоминать в декларации DIMENSION. Однако можно имя массива без индексов указать в декларации типа, а его размерности задать в декларации DIMENSION.
5. Декларации типа переменных относятся к классу неисполняемых инструкций и должны предшествовать в программе всем исполняемым инструкциям.
6. *Подразумеваемый (по умолчанию) тип:* если переменная не описана в декларациях типа, то тип такой переменной определяется по первой букве ее имени: переменные, имена которых начинаются с букв I, J, K, L, M или N, считаются целыми, все остальные неописанные переменные — вещественными.

Другим средством, с помощью которого может быть объявлен тип переменных, служит инструкция IMPLICIT — отсутствующая в ряде диалектов. В инструкции IMPLICIT указывается тип и диапазон букв. В отсутствие других деклараций типа переменных все переменные, имена которых начинаются с букв в указанном диапазоне, будут относиться к этому типу. Неявно определенный тип переменной может быть переопределен для отдельных переменных посредством деклараций REAL, INTEGER или LOGICAL. Декларация IMPLICIT имеет следующий вид:

IMPLICIT *тип* (*диапазон* 1 [, *диапазон* 2,...])

где *тип* — это какой-либо тип, т. е. REAL, INTEGER или LOGICAL.

Каждый *диапазон* — это или одиночные буквы, или пары букв, разделенные дефисом.

Пример:

IMPLICIT INTEGER (C — D, H — K, Z)

Действие: Любые переменные, имена которых начинаются с букв C, D, H, I, J, K или Z, объявляются целыми переменными.

Примечания:

1. Декларация IMPLICIT должна предшествовать в программе любым другим декларациям.
2. Подразумеваемые (по умолчанию) типы переменных могут быть изменены для отдельных переменных с помощью явных деклараций REAL, INTEGER или LOGICAL.
3. Инструкция IMPLICIT расширяет диапазон неявного описания переменных по первым буквам. В приведенном

выше примере переменные, начинающиеся с букв L, M или N, будут целыми по умолчанию (переменные, начинающиеся с букв в диапазоне с I по N, считаются целыми при отсутствии явных деклараций).

Обычно, когда ячейка памяти отводится для переменной, содержимое этой ячейки неизвестно, и поэтому мы считаем, что переменная не определена программистом. Однако мы можем определить начальное значение, присваиваемое переменной, с помощью инструкции DATA.

DATA *спис-перем* 1 /*значения* 1/ [,*спис-перем* 2/ *значения* 2/,...]
Здесь каждый *спис-перем* — это список простых переменных или элементов массива; в некоторых диалектах Фортрана в списке допускается также использование массивов.

● *Значения* — это константа или последовательность констант, разделенных запятыми. Эти значения присваиваются переменным в предшествующем *спис-перем*. Запись *n*значение*, где *n* — некоторая целая константа, означает, что указанное значение должно быть повторено *n* раз ●

Пример:

```
INTEGER TOP, K, L
REAL X(3, 2), Y (10, 20)
DATA TOP /5/, X/3*1., 3*10./, K, L/0. 1/, Y(5, 2) /13/
```

Действие: Инструкция DATA присваивает переменной TOP значение 5; элементам массива X значения 1., 1., 1., 10., 10., 10.; переменным K и L значения 0 и 1 соответственно; элементу Y(5, 2) значение 13.

Примечания:

1. Если в списке переменных инструкции DATA указано имя массива, то подразумеваются сразу все элементы массива.
2. Должно быть соответствие один к одному между элементами списка и константами, указанными в списке значений.
3. Соответствие между переменными и их значениями устанавливается порядком их записи слева направо. Если в списке переменных указано имя массива, то его индивидуальным элементам присваиваются значения в порядке расположения элементов массива в памяти (т. е. по столбцам).
4. ● Тип каждой переменной и связанное с переменной начальное значение должны соответствовать друг другу, за исключением литерных констант, которые могут быть присвоены переменным любого типа ●

5. Инструкция DATA должна следовать в программе после всех деклараций размерности и типа.
6. Инструкция присваивает только начальные значения перед выполнением программы. Если в ходе выполнения значение переменной изменяется, оно не может быть восстановлено посредством инструкции DATA.
7. В программе можно использовать несколько инструкций DATA.
8. Начальное значение любой переменной, которая не упоминалась в инструкции DATA, не определено.

В некоторых реализациях Фортрана установку начальных значений данных можно делать в декларациях типа переменных. В этом случае тип, размерность и начальные значения переменных могут быть объявлены с помощью одной декларации. Например,

```
INTEGER I/0/, LIST (5, 2) /5*0,5*1/
```

объявляет I и LIST целыми. I — простая переменная с начальным значением 0; в массиве LIST размером 5*2 первому столбцу присвоены начальные значения 0, второму столбцу — единицы.

0.3.2. Инструкция END

Инструкция END указывает на окончание последовательности инструкций главной программы или подпрограммы. Она записывается как

```
END
```

т. е. как одиночное END, и должна физически быть последней инструкцией программы.

Пример:

```
END
```

Действие: Завершает компиляцию главной программы или подпрограммы.

Примечания:

1. В программе или подпрограмме должна быть только одна инструкция END.
2. Инструкция END должна быть последней в каждой программе или подпрограмме.
3. Инструкция неисполняемая, у нее не может быть метки.

На этом мы завершаем обзор некоторых основных неисполняемых инструкций Фортрана. Инструкция FORMAT будет объяснена в разд. 0.5.4, а другие неисполняемые инструкции —

в последующих главах. Мы перейдем теперь к изучению исполняемых инструкций Фортрана. Вначале рассмотрим, как присваивать значения переменным и выполнять ввод и вывод; затем в разд. 0.6 опишем инструкции локального управления последовательностью действий. И наконец, в разд. 0.7 перейдем к изучению подпрограмм Фортрана.

0.4. Инструкция присваивания

Инструкция присваивания заменяет старое значение переменной на значение некоторого выражения. Инструкция Фортрана A=B означает, что значение переменной B пересылается в ячейку памяти, занимаемую переменной A. Эта операция интерпретируется как «присвоить данное значение B переменной A», а вовсе не «A равно B». Так, последовательность инструкций

```
A=5
B=8
A=B
```

присваивает переменным A и B значение 8. Старое значение A (5 для данного примера) теряется. Инструкция присваивания имеет следующие формы записи:

[*метка*] *имя-арифм-перем.=арифм-выраж.*

или

[*метка*] *имя-логич-перем.=логич-выраж.*

где *метка* — это номер инструкции;

имя-арифм-перем. — имя целой или вещественной простой переменной;
имя-логич-перем. — имя простой логической переменной или элемент массива;
арифм-выраж. — арифметическое выражение;
логич-выраж. — логическое выражение.

Пример:

```
10 KOUNT=2.7*(I—1)
   LIST(3)=LIST(3)+1
20 ON=LIST(I).LT.KOUNT
   VALUE=LIST(KOUNT)**I/2 — LIST(I)
30 LIST(2*I—1)=LIST(KOUNT—2)*LIST(I)
```

Действие:

В каждой инструкции присваивания вычисляется выражение, стоящее справа от знака равенства, и это значение заменяет значение переменной в левой части инструкции.

Примечания:

1. К моменту выполнения инструкции присваивания каждая переменная, находящаяся в правой части, должна иметь соответствующее значение.
2. Тип результата вычисления выражения (целый или вещественный) в *арифм-выраж.* преобразуется к типу *имя-арифм-перем.*, если они отличны друг от друга.
3. В левой части инструкции присваивания может быть только одна переменная.

Обратимся к инструкциям присваивания, приведенным в примере. Пусть к моменту выполнения инструкции 10 переменные имели следующие значения:

KOUNT	14	целое
LIST(3)	6	целое
LIST(5)	4	целое
ON	.TRUE.	логическое
VALUE	15.3	вещественное
I	3	целое

В инструкции с меткой 10 вычисление выражения 2.7*2 дает 5.4, и это значение должно быть присвоено переменной KOUNT. Но так как KOUNT должна быть целой, усечение дробной части приводит к тому, что переменной KOUNT присваивается значение 5 (см. 2-е примечание). В следующей инструкции 6+1, т. е. 7, присваивается элементу LIST(3). Таким образом, LIST(3) изменяет свое значение с 6 на 7. В инструкции с меткой 20 вычисление логического выражения в правой части дает результат .FALSE., так как значение KOUNT равно теперь 5, а LIST(3) имеет значение 7. Поэтому значение .FALSE. присваивается переменной ON. В следующей инструкции вычисляется выражение ((4**3)/2)—7=25. Но так как VALUE — вещественная переменная, ей присваивается значение 25.0. В инструкции с меткой 30 элементу массива LIST(5) присваивается значение 7*7=49.

После выполнения пяти инструкций присваивания простые переменные и элементы массива принимают следующие значения:

KOUNT	5	ON	.FALSE.
LIST(3)	7	VALUE	25.0
LIST(5)	49	I	3

0.5. Инструкции ввода/вывода

В этом разделе мы рассмотрим инструкции ввода и вывода, или, сокращенно, инструкции В/В. Значения данных, вводимых инструкцией ввода READ, присваиваются переменным, список

которых программист указывает в самой инструкции READ. С помощью инструкций вывода WRITE и PRINT можно, например, выводить текущие значения переменных на устройство печати.

• В некоторых диалектах Фортрана (например, WATFIV) допустим бесформатный ввод/вывод. В этом случае программисту не нужно точно задавать позиции данных в записи. При форматном вводе/выводе с помощью формата программист точно определяет форму каждой записи •

Форматным В/В можно пользоваться во всех реализациях Фортрана.

0.5.1. Списки переменных для ввода и вывода

Начнем со структуры списка переменных в инструкциях ввода и вывода.

в-в-список может обозначать:

1) *спис-перем*
2) *в-в-список*, *в-в-список*
3) (*в-в-список*, *цел-перем-нач*, *конечн* [,*приращ*])

где *спис-перем* — список простых переменных, элементов массива, имен массивов.

цел-перем — неиндексированная целая переменная. *нач*, *конечн* и *приращ* — целые константы или простые целые переменные, значение каждой из них больше нуля. *По умолчанию* принято приращение (если оно опущено), равное 1.

Простая переменная или элемент массива в *в-в-списке* указывает, что вводится или выводится одиночная величина. Если же в *в-в-списке* задано имя массива без индекса, то оно представляет *все* элементы массива в памяти (упорядоченные по столбцам). В третьей форме *в-в-списка* используется конструкция *неявного цикла*, сходная с инструкцией цикла DO. (Инструкция DO подробно описана в разд. 0.6.4.) Эта форма *в-в-списка* особенно удобна при составлении списков переменных и элементов массивов данных. В неявном цикле элементы *в-в-списка* генерируются с помощью *цел-перем*, которой вначале присваивается *начальное* значение, и затем при каждом повторении оно увеличивается на *шаг приращения*, пока не достигнет конечного значения. Так, неявный цикл (Y(2*I—1), X (I), I=3, 8, 2) компилятор Фортрана заменит списком Y (5), X (3), Y (9), X (5), Y (13), X (7). Значение *цел-перем* I, равное 9, превышает конечное значение индекса. Поэтому компилятор «вычисляет» (Y (2*I—1), X (I)) только для I, равного 3, 5, 7, и в результате создает список из 6 переменных.

Пример

	Список:	*Представляет:*
(а)	SUM, TOP(I), J	SUM, TOP(I), J
(б)	(X(I), I = 3,5)	X(3), X(4), X(5)
(в)	((Y(K,J), K = 1,5), J = 1,M)	Y(1,1), Y(2,1), Y(3,1), ..., Y(5,1), Y(1,2), Y(2,2), ..., Y(5,M)

0.5.2. Бесформатный ввод

Бесформатный ввод, допускаемый диалектом WATFIV и другими диалектами Фортрана, особенно удобен для начинающих программистов. При бесформатном вводе не нужно указывать колонки перфокарты, в которых расположены данные. Данные располагаются в любых позициях перфокарты, элементы данных отделяются один от другого пробелами и/или запятыми. Приведем структуру инструкции бесформатного ввода READ:

[*метка*] READ, *в-в-список*

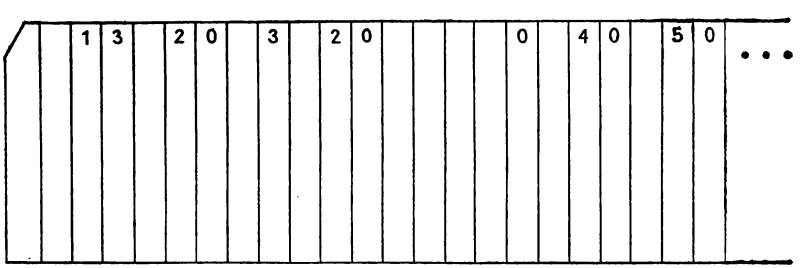
или

[*метка*] READ (*устр-ввода*,*[,END-*выход*]) *в-в-список*

где *метка* и *в-в-список* были описаны выше;
устр-ввода — целая константа или переменная, указывающая номер устройства ввода;
* означает, что данные вводятся без формата;
выход — номер (метка) исполняемой инструкции.

Примеры

(а) READ, I, TOP, (LIST (J), J=1, 5, 2)
(б) READ (5, *, END=99) I, TOP, (LIST(J), J=1, 5, 2)



Карта с данными

Действие: (для приведенных примеров): ввод пяти чисел с перфокарты и присваивание их по порядку переменным списка:

I	TOP	LIST(1)	LIST(3)	LIST(5)
13	20.3	20	0	40

Примечания:

1. При каждом исполнении инструкции READ вводится одна перфокарта.
2. Данные при вводе без формата могут располагаться в любом месте перфокарты и отделяются друг от друга запятыми и/или пробелами.
3. Каждое входное значение должно согласовываться по типу с переменной, которой оно присваивается при вводе.
4. Инструкция READ вводит лишь столько значений, сколько их указано в *в-в-списке*, остальные данные игнорируются. Если требуется ввести больше чисел, чем помещается на одной перфокарте, вводится следующая карта, и так до тех пор, пока не будут выбраны все элементы данных, указанные в *в-в-списке*, или пока не будут исчерпаны все перфокарты.
5. Значение *устр-ввода* зависит от реализации языка. Для устройства ввода с перфокарт машин IBM 360-370 с компилятором WATVIF *устр-ввода*=5.
6. Когда считаны все перфокарты, а инструкция READ требует ввода, регистрируется ошибка В/В и выполнение программы прекращается, если не задано значение *выход* для END. END=*выход* передает управление на инструкцию с меткой *выход*.

0.5.3. Бесформатный вывод

Вывод без формата осуществляет инструкция PRINT. Подобно бесформатной инструкции READ, эта инструкция полезна для начинающих программистов, так как при ее использовании не нужно точно задавать позиции в печатаемой строке. Инструкция PRINT также очень полезна при отладке программ, поскольку позволяет выводить данные на отдельных стадиях вычислений. Инструкция PRINT имеет вид

- [*метка*] PRINT, *выв-список*

где *метка* — метка инструкции, а *выв-список* — список, имеющий ту же форму, что и *в-в-список*, но, кроме простых переменных, элементов и имен массивов, он может содержать константы и выражения •

Пример

```
    PRINT, I, (X(J), J=1, 2), SUM, '=TOTAL'
где I=3  X(1)=30.  X(2)=87.2  SUM=117.2
```

Действие: печатается такая строка:

3 0.300000 E 02 0.872000 E 02 0.1172000 E 03=TOTAL

Примечания:

1. При выполнении каждой инструкции PRINT печать на АЦПУ начинается следующей по порядку строки.
2. Если для вывода значений из *выв-списка* требуется более одной строки, печать автоматически переходит на следующую строку.
3. Целые числа выводятся на печать, выравненные по правому краю (в столбцах); нули, предшествующие значащей цифре, подавляются.
4. Вещественные числа обычно печатаются в формате E (порядок-мантисса).
5. Все величины выводятся в порядке их расположения в *выв-списке*.

0.5.4. Инструкция FORMAT и форматный ввод/вывод

Как уже говорилось, бесформатный ввод/вывод удобен для начинающих программистов. Однако при программировании сложных задач очень важно уметь располагать данные в определенных позициях печатной страницы, точно задавать позиции входных данных на перфокарте для считывания их по программе. При вводе некоторые колонки перфокарты иногда нужно игнорировать, а при выводе — разделять данные пробелами. Для того чтобы это сделать, необходимо точное управление форматами ввода и вывода данных.

При форматном вводе/выводе данные точно располагаются в соответствии с *форматом*. Запись — это последовательность полей, и инструкция FORMAT попросту задает структуру полей в записи. Инструкция FORMAT неисполняемая, она определяет позицию, размер и тип данных в записи. Каждое поле содержит значение отдельного элемента данных. Инструкция FORMAT имеет следующий вид:

метка FORMAT *(формат список)*

Здесь *метка* — обязательный элемент инструкции;

формат список — последовательность из одного или более кодов формата, разделенных запятыми. Основные коды формата [1]):

---

[1]) С остальными форматами можно познакомиться в руководствах по Фортрану.

Ниже используются такие обозначения: n, w и d должны быть целыми константами без знака; n можно опустить, в этом случае предполагается, что $n=1$.

nIw : n целых чисел с шириной поля w.

nF$w.d$: n десятичных (вещественных) чисел с шириной поля w, d — число цифр после десятичной точки.

nE$w.d$: n вещественных чисел с порядком, ширина w, d — число цифр после десятичной точки.

nAw : n буквенно-цифровых цепочек, каждая из которых содержит w литер.

wH *цепочка-из-w-литер*: литеральное [2]) поле, состоящее из w литер, непосредственно следующих за H.

'любая-цепочка': литеральное поле (допускается не во всех диалектах Фортрана).

nX : n игнорируемых позиций во входной записи или n пробелов на выводе.

Tw : пропуск до колонки с номером w (принят не во всех диалектах Фортрана).

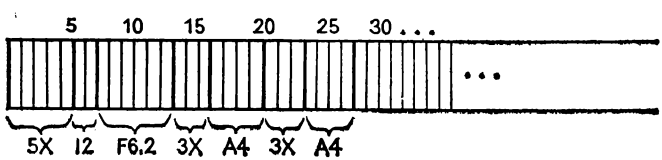
/: конец текущей записи (строки или перфокарты).

n *(формат список)*: n — кратное повторение группы кодов формата, заключенной в скобки.

Пример:

```
100 FORMAT (5X, I2, F6.2, 2(3X, A4))
```

Действие: Инструкция FORMAT определяет семь начальных полей записи, как показано на рисунке.



Примечания:

1. Инструкция FORMAT не обязательно должна описывать всю запись целиком.
2. К инструкции FORMAT можно обращаться в различных инструкциях ввода/вывода.
3. Поля литеральных констант (спецификация H и цепочки, заключенные в кавычки) обычно используют только для вывода информации.

---

[2]) Речь идет о литералах Фортрана, т. е. о величинах, заданных своим значением; их не следует путать с «литерами», т. е. буквенно-цифровыми символами. Литералом может быть числовая константа или цепочка литер.— *Прим. ред.*

4. Инструкция FORMAT может быть помещена в любом месте программы. Эта инструкция неисполняемая.
5. Если все коды формата в инструкции FORMAT выбраны, но еще остались переменные в соответствующих списках инструкций READ или WRITE, то начинается обработка следующей записи. При этом повторно используется группа кодов формата, относящаяся к последней закрывающейся скобке (но не к скобке, которая закрывает весь список форматов). Если же вложенной повторяемой группы кодов нет, то повторно используется весь формат.

0.5.5. Форматный ввод

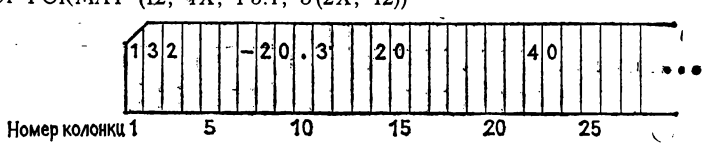
Для форматного ввода используется инструкция READ, которая в этом случае имеет следующий вид:

[*метка*] READ (*устр-ввода*, *формат* [, END-*выход*]) *в-в-список*

где: *метка* — метка инструкции;
устр-ввода — целое число, указывающее номер устройства ввода;
формат — метка соответствующей инструкции FORMAT;
выход — метка исполняемой инструкции (не во всех диалектах можно использовать подпараметр END=);
в-в-список —список, определение которого дано в разд. 0.5.1.

Пример:

```
    READ (5, 101, END=50) I, TOP, (LIST(J), J=1, 5, 2)
101 FORMAT (I2, 4X, F5.1, 3(2X, I2))
```



Действие: Ввести с перфокарты пять значений в соответствии с инструкцией FORMAT с меткой 101. В том случае, если встретится признак конца данных, управление передать на инструкцию с меткой 50. В данном примере переменным в списке *в-в-список* будут присвоены следующие значения:

I	13
TOP	—20.3
LIST(1)	20
LIST(3)	0
LIST(5)	40

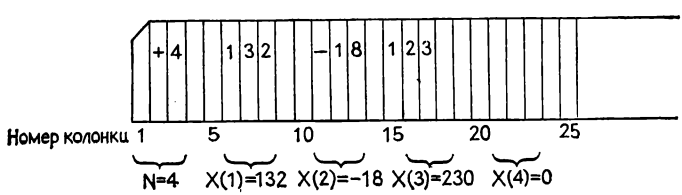
Примечания:

1. При каждом выполнении инструкции READ будет вводиться новая запись.

2. Значения присваиваются переменным списка в соответствии с инструкцией FORMAT, метка которой указана в данной инструкции READ.
3. Каждое вводимое значение должно быть согласовано с кодом формата и с типом переменной, которой оно присваивается. Исключение здесь составляет только код формата A. В этом случае вводимые значения могут присваиваться любым переменным.
4. Пробелы в числовых полях интерпретируются как нули.
5. Значение *устр-ввода* зависит от реализации. Для машин IBM 360-370 *устр-ввода*=5 — устройство ввода с перфокарт; для машин DEC-10 *устр-ввода*=2 — устройство ввода с перфокарт и *устр-ввода*=5 — ввод с терминала.

Рассмотрим теперь два примера использования инструкции форматного ввода READ для того, чтобы проиллюстрировать ее связь с инструкцией FORMAT при вводе данных с перфокарт.
Пример 1: *Ввод целых чисел*

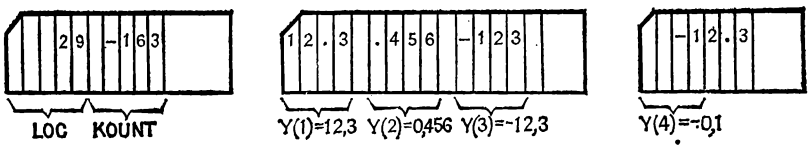
```
      INTEGER LIST(50), N
      READ (5,100) N, (LIST(I),I=1,N)
  100 FORMAT (16(I3, 2X))
```



В этом примере первое число на перфокарте (4) присваивается переменной N. Последующие числа присваиваются первым N элементам массива LIST. Заметим, что знак входит как часть поля, отводимого для N. Так как пробелы при вводе соответствуют нулям, целые числа должны занимать строго определенные позиции на перфокарте. В инструкции FORMAT содержится больше полей (16 целых полей), чем на карте с данными. Это не будет ошибкой. При выполнении инструкции READ не обязательно использовать все поля инструкции FORMAT. Формат обычно записывают с учетом максимального числа элементов данных, которые могут быть в записи; например, для перфокарты можно записать 10I8 или 8I10.

Пример 2: *Ввод данных разных типов*

```
      REAL Y(10)
      INTEGER LOC, KOUNT
      READ (5,101) LOC, KOUNT, (Y(I), I=1, 4)
101   FORMAT (2I5 / 3(F4.1,1X))
```



В этом случае будут считаны три перфокарты, так как формат описывает только пять полей, а затребовано шесть чисел. С первой перфокарты вводятся значения переменных LOC и KOUNT. Разделитель / указывает на окончание первой записи (перфокарты). Далее по формату F4.1 будут считаны значения со второй карты. После чтения второй карты вновь используется последняя повторяемая группа кодов формата для считывания следующей перфокарты. Последнее поле F4.1 вводится с третьей карты. Заметим, что десятичная точка, находящаяся в какой-либо позиции поля, остается в этой же позиции и во введенной величине (реальное положение точки имеет приоритет по сравнению с тем, что указано в инструкции FORMAT). Число без десятичной точки будет преобразовано в соответствии со спецификацией формата (будет выставлена десятичная точка «на расстоянии» *d* цифр от правого конца числа). Знак и десятичная точка, если они пробиты на перфокарте, занимают по одной позиции.

Советуем вам использовать подпараметр END. Он позволяет сигнализировать о завершении отдельной фазы выполнения программы. Если признак конца данных оказался неожиданным, можно вывести сообщение вместе со значениями наиболее важных переменных и остановить выполнение программы.

0.5.6. Форматный вывод

В этом разделе мы дадим определение форматного вывода, который выполняется с помощью инструкции WRITE.

[*метка*] WRITE (*устр-выв*, *формат*) [*в-в-список*]

где: *метка* — номер инструкции;
устр-выв — целая константа или переменная, определяющая устройство вывода;
формат — метка соответствующей инструкции FORMAT;
• *в-в-список* — необязательный элемент инструкции •

Пример: Пусть RMAX и SUM — вещественные переменные, значения которых —13.2 и 183.765 соответственно.

```
      WRITE (6, 100) RMAX, SUM
  100 FORMAT ('RMAX=', F7.2, 5X, 'TOTAL IS$', F6.2)
```

Действие: Выводится следующая строка:

```
RMAX=—13.20  TOTAL  IS  $183.77
```

Примечания:

1. Если *устр-выв* определяет АЦПУ, первая литера выводимой на печать строки управляет подачей бумаги. Эта литера называется *символом управления* и не печатается. К символам управления относятся: '+' печатать на той же самой строке, ' ' пропустить одну строку, '0' пропустить две строки и '1' перейти на новую страницу.
2. Значение *устр-выв* зависит от реализации. Для машин IBM 360-370 *устр-выв*=6 — это АЦПУ; для машин DEC-10 *устр-выв*=3 — это АЦПУ и *устр-выв*=5 — вывод на терминал.
3. Тип каждой выводимой переменной должен быть согласован с кодом соответствующего формата, за исключением кода A. В формате A могут быть выведены данные любого типа.
4. Вещественные числа округляются, если соответствующие поля в формате имеют меньше цифр после десятичной точки, чем в представлении чисел.
5. Каждому элементу данных должно соответствовать поле в формате.
6. Знак минус (и десятичная точка в полях типа F или E) засчитывается в ширине *w*.
7. С помощью формата не может быть выдана на печать запись длиннее максимального размера, принятого для данного печатающего устройства. Типичные максимальные размеры выводимых на печать записей — 121 или 133 литеры (длина строки). На терминалы можно выводить записи длиной 72 или 80 литер. На перфокарте обычно располагается не более 80 литер. В формате, конечно, может быть определено несколько записей.
8. Код *n*X вызывает появление в выводимой записи *n* пробелов.
9. Литерные данные (коды A*w*, *n*H или литералы) выравниваются в выходной записи по левому краю поля; целые и вещественные величины выравниваются по правому краю поля (нули, предшествующие первой значащей цифре, не печатаются).

Приведенный ниже пример показывает типичное применение инструкции FORMAT для вывода. В этой инструкции FORMAT используется косая черта, которая служит для указания конца

записи, и группа повторений внутри формата. Предположим, что мы хотели бы вывести на печать, начиная с новой страницы, информацию, расположенную следующим образом:

```
                    * * SUMMARY REPORT * *

  # BOUGHT:  53      # SOLD: 138        PROFIT: $1390.50

      PART #          STOCK      TOTAL =   141
      ------          -----
        1               13
        2               28
        3                0
        4               50
        5               18
        6               32
```

где BOUGHT, SOLD и TOTAL — простые целые переменные, STOCK — массив целых чисел размера N и PROFIT — простая вещественная переменная.

```
    WRITE (6,100) BOUGHT, SOLD, PROFIT, TOTAL, (I, STOCK(I), I=1,N)
100 FORMAT ('1',15X, '* * SUMMARY REPORT * *', //
  1  5X,'# BOUGHT:', I4, 5X,'# SOLD:', I4, 7X,'PROFIT: $', F7.2//
  2   T12, 'PART #', 10X, 'STOCK', 5X 'TOTAL = ', I6/
  3   T12, '------', 10X, '-----'/
  4  (T12, I4, 2X,   10X, I4) )
```

FORMAT с меткой 100 определяет 12 записей, которые выводятся на печать в 10 строках. Последняя группа формата определяет только два возможных поля (код I4), и поэтому она используется шесть раз для вывода 12 величин. Каждый раз поле T12 вызывает перемещение на одну строку, так как первый символ в выводимой на печать строке (он не печатается) — это знак пробела.

На этом мы заканчиваем обсуждение инструкций ввода/вывода Фортрана с использованием перфокарт, терминала и печатающего устройства. В гл. 4 вы познакомитесь с другими устройствами ввода/вывода, такими, как магнитные ленты и магнитные диски, вы узнаете также о формах представления данных на этих устройствах.

0.6. Инструкции локального управления последовательностью действий

Программист, разумеется, может управлять порядком выполнения инструкций в программе. Обычно инструкции выполняются последовательно одна за другой. Программист может изменить этот порядок с помощью инструкций прекращения выполнения (STOP), безусловного перехода (GO TO), условного перехода

(IF) и инструкции цикла (DO). В данном разделе мы рассмотрим эти инструкции, которые позволяют программисту управлять последовательностью выполнения инструкций.

0.6.1. Прекращение выполнения программы

Выполнение программы может быть остановлено в любое время с помощью инструкции STOP:

[*метка*] STOP

где *метка* — номер инструкции.

Пример:

```
15 STOP
```

Действие: Выполнение этой инструкции в любом месте программы или подпрограммы вызывает прекращение работы программы.

Примечание: В программе может быть произвольное число инструкций STOP.

0.6.2. Безусловный переход

Команды передачи управления Фортрана дают возможность изменять естественную последовательность выполнения инструкций. Здесь мы приведем две инструкции, которые осуществляют безусловную передачу управления. Наиболее простая форма безусловного перехода в Фортране — это инструкция GO TO следующего вида:

[*метка* 1] GO TO *метка* 2

где *метка* 1 и *метка* 2— номера инструкций.

Пример:

```
GO TO 5
```

Действие: Управление передается на инструкцию с меткой 5.

Примечания:

1. *метка* 2 должна быть номером (меткой) исполняемой инструкции в той же главной программе или подпрограмме, в которой находится инструкция GO TO.
2. Слова GO TO и метку в инструкции для удобства чтения и во избежание путаницы следует разделять пробелами. Некоторые замечания о структуре программы и свойствах использования инструкции GO TO приведены в гл. 1.

Более гибким средством безусловной передачи управления является *вычисляемый переход GO TO*. Во время выполнения программы значение целой переменной определяет, на какую метку из списка меток надо передать управление. Вычисляемый GO TO имеет следующий вид:

[*метка* 0] GO TO (*метка* 1, *метка* 2, . . ., *метка n*), *цел-перем*

где: *метка* 0 — номер инструкции GO TO; *метка* 1, *метка* 2 и т. д.— номера исполняемых инструкций в той же главной программе или подпрограмме, что и инструкция GO TO;
• *цел-перем* — простая целая переменная •

Пример:

GO TO (10, 2, 15, 15), K

Действие: Если K=1, перейти к инструкции с номером 10; при K=2— к инструкции с номером 2; при K=3 или 4— к инструкции с номером 15.

Примечание. Если вычисляемый GO TO содержит список из *n* меток, то величина *цел-перем* должна лежать в интервале от 1 до *n*.

0.6.3. Условные переходы

Инструкции условного перехода дают возможность программисту выбирать условия, в соответствии с которыми происходит передача управления на ту или иную инструкцию программы. Существует две формы инструкций IF условного перехода Фортрана: арифметический и логический переходы. Арифметический условный переход IF содержит арифметическое условие, на основании которого происходит передача управления. Инструкция имеет вид

[*метка*] IF (*арифм-выр*) *отриц*, *нулев*, *полож*

где *метка* — метка инструкции IF;
арифм-выр — арифметическое выражение;
отриц, *нулев* и *полож* — метки инструкций, на которые передается управление в зависимости от значения *арифм-выр*.

Пример:

IF (X/2—A) 3, 5, 4

Действие: Вычислить (X/2—A) и передать управление инструкции с меткой 3, если результат вычисления арифметического выражения отрицательный; инструкции с меткой 5, если результат нулевой; инструкции с меткой 4, если он положительный.

Хотя арифметическое выражение в инструкции IF может быть сколь угодно сложным, обычно используют простые выражения. Допускаются арифметические выражения любого типа; однако следует быть осторожным при применении выражений вещественного типа. В результате ошибки округления вещественной величины она может оказаться не равной в точности нулю, даже если истинное значение *арифм-выр* равно нулю. В связи с этим может произойти передача управления на метки *полож* или *отриц*, в то время как ожидалась передача управления на метку *нулев*.

Другая инструкция условного перехода — логическая инструкция IF. Она позволяет установить более сложные условия, чем арифметическая инструкция IF, но может передать управления только по двум направлениям, в то время как арифметическая инструкция IF может передавать управление по трем направлениям. (Надо сказать, однако, что передача управления по трем направлениям в программировании используется сравнительно редко.) Логическую инструкцию IF не обязательно применять для передачи управления: она может вызывать считывание перфокарты, печать строки, присваивания или останов. Основное преимущество логической инструкции IF по сравнению с арифметической состоит в том, что она более наглядна.

[*метка*] IF (*логич-выр*) *инстр*

где *метка* — номер инструкции IF;
логич-выр — логическое выражение;
инстр — любая исполняемая инструкция (именно инструкция, а не метка), кроме арифметического условного перехода IF или инструкции цикла DO.

Пример:

```
IF (A .GT. 10.0) GO TO 32
```

Действие: Если A больше 10.0, передать управление инструкции с меткой 32; в противном случае выполнить следующую по порядку инструкцию.

Примечания:

1. Если значение *логич-выр* .TRUE., то выполняется *инстр*.
2. Если *инстр* не GO TO, STOP или RETURN, то вслед за *инстр* будет выполняться инструкция, непосредственно следующая за IF. В том случае, если не соблюдено условие для выполнения *инстр* (т. е. значение *логич-выр*=.FALSE.), управление сразу будет передано на инструкцию, непосредственно следующую за инструкцией IF.

Приведем еще один пример использования логической инструкции IF. Допустим, что требуется заменить каждый элемент

n-элементного массива X его абсолютным значением и затем распечатать весь массив.

```
      I = 1
2     IF (X(I) .LT. 0.) X(I) = -X(I)
        PRINT, X(I)
        I = I+1
      IF (I .LE. N) GO TO 2
```

Заметим, что независимо от значения (истина или ложь) логического выражения в инструкции с меткой 2 PRINT будет выполняться для каждого значения I в интервале от 1 по N.

0.6.4. Инструкции цикла

Фортран располагает мощным средством, с помощью которого можно несколько раз повторить ту или иную последовательность инструкций. Так как повторение или «циклы» в программировании применяются очень широко, существует инструкция, специально предназначенная для организации циклов. Это инструкция DO. Она имеет вид

[*метка*] DO *инстр цел-пер* = *нач-зн*, *кон-зн* [,*шаг*]

где *метка* — номер данной инструкции;

инстр — номер (метка) исполняемой инструкции, на которой заканчивается цикл DO;

цел-пер — простая целая переменная, называемая индексом цикла DO;

• *нач-зн*, *кон-зн* и *шаг* — также простые целые переменные или целые константы, каждая из которых больше нуля •

Пример:

DO 32 I=1, 10, 2

Действие: Выполнить инструкции, расположенные от инструкции DO по инструкцию с меткой 32 включительно пять раз со значениями I, равными 1, 3, 5, 7, 9.

Примечания:

1. Цикл DO выполняется как минимум один раз.
• 2. Значения *цел-пер*, *нач-зн*, *кон-зн* и *шага* не должны изменяться программой внутри цикла DO •
3. Из цикла DO может быть передано управление. Управление можно также передать от инструкции внутри цикла к другой инструкции внутри цикла; однако от инструкции, расположенной вне цикла DO, может быть передано управление только в начало цикла. Передавать управление извне внутрь цикла DO не разрешается.

4. Роль инструкции с меткой *инстр* не могут играть: STOP, GO TO, RETURN (см. разд. 0.7.4), логическая IF в сочетании с любой из этих инструкций, инструкция DO или арифметическая инструкция IF.
5. Циклы DO могут быть вложенными один в другой, т. е. все инструкции каждого внутреннего цикла должны укладываться внутри внешнего цикла. (Два или более цикла DO могут иметь одну и ту же метку *инстр* последней выполняемой инструкции.)
6. *По умолчанию* принято, т. е. если *шаг* опущен, что шаг равен 1.

Возможно, лучший способ объяснить цикл DO — это продемонстрировать его работу на примере. На рис. 0.7 показан пример цикла DO и эквивалентная структура без использования цикла DO.

```
  DO 32 I = 1,10,2
  TOP = TOP + I
        :
        :
  X(I) = X(I) + 1
32 PRINT, X(I)
```

```
      I = 1                <==НАЧ-ЗНАЧ
5     TOP = TOP + I
        :
        :
      X(I) = X(I) + 1
      PRINT, X(I)
      I = I+2              <==ШАГ
      IF (I.LE.10) GO TO 5 <==ПРОВЕР-УСЛ
```

Рис.0.7. Действие инструкции DO.

Перейдем теперь к изучению инструкции CONTINUE. Эта инструкция может иметь метку. Сама по себе она не выполняет каких-либо действий. Чаще всего CONTINUE — последняя инструкция *(инстр)* цикла.

[*метка*] CONTINUE

где *метка* — номер данной инструкции.

Пример:

20 CONTINUE

Действие: Никакого; эта инструкция не вызывает никаких действий. Она содержит метку 20, на которую можно передавать управление.

Примечание: Чаще всего используется в качестве последней в цикле DO. Встретив CONTINUE в такой роли, компилятор генерирует команды приращения и проверки индекса цикла. Выполнение инструкции CONTINUE как бы приводит к приращению счетчика цикла и к очередному выполнению тела цикла.

Перейдем теперь к рассмотрению более сложного примера с циклом DO. Пусть два линейных массива (вектора) имеют размеры N и M соответственно. Наша задача — напечатать каждый положительный элемент массива X, который содержится в массиве Y, и список ячеек памяти, в которых находятся значения X, появляющиеся также и в Y.

```
   DO 10 I = 1, N
      IF (X(I) .LE. 0.) GO TO 10
         DO 20 J = 1, M
20       IF (X(I) .EQ. Y(J)) PRINT, 'MATCH', X(I), 'AT J=', J
10 CONTINUE
```

(Здесь 'MATCH' — текст, означающий «соответствует», и 'AT J=' означает «при J=».)

В данном примере инструкция CONTINUE с меткой 10 — последняя инструкция цикла. Мы не можем закончить цикл предыдущей инструкцией, так как, если элемент массива X меньше или равен нулю, его необходимо игнорировать и нужно просматривать следующий. Поэтому нам следует пропустить второй цикл полностью. Заметим, что во внутреннем цикле используется другой индекс (J), поскольку внешний индекс (I) не может быть изменен во внутреннем цикле. Внутренний цикл можно также записать в виде

```
   DO 20 J = 1,M
      IF (X(I) .EQ. Y(J)) PRINT, 'MATCH', X(I), 'AT J=', J
20 CONTINUE
```

Применение инструкции CONTINUE для окончания цикла DO предпочтительно, поскольку она точно отмечает конец цикла. В дальнейшем мы всегда будем использовать ее в этой роли.

В последнем примере данного раздела показано, как можно использовать значение индекса после выхода из цикла DO. Предположим, что мы хотели бы ввести перфокарты с данными и затем остановиться, когда перфокарты окончились или когда было считано 100 перфокарт. Данные, считанные с перфокарт, заносятся в массив X, и затем, используя переменную NCARDS — фактическое число считанных перфокарт, выведем на печать число элементов массива X, превышающих значение X (NCARDS). Полная программа, выполняющая это задание показана на рис. 0.8.

Программа поясняется комментариями, вставленными в тело каждого цикла DO. Как мы установили ранее, инструкции FORMAT могут быть расположены в любом месте программы. В данном примере мы решили поместить все инструкции FORMAT непосредственно перед инструкцией END. Параметр END=*метка* в инструкции READ передает управление инструкции с меткой 15 в том случае, если введены все перфокарты. Индекс NCARDS

необходимо уменьшить на 1 для указания действительного числа элементов массива, определенного в цикле ввода. Значение индекса DO уже не потребуется, если завершилось требуемое число повторений, т. е. цикл окончился нормально. В этом случае окончательное значение индекса зависит от компилятора. Наконец,

```
C. . . . . .ПРОГРАММА ВЫВОДА НЕКОТОРЫХ ЗНАЧЕНИЙ . . . . . . . .
C. . . . . .ПРЕВЫШАЮЩИХ ПОСЛЕДНЕЕ ВВЕДЕННОЕ . . . . . . . . . .
      INTEGER X(100), NCARDS, IN, OUT, FINAL, NCM1, COUNT
      DATA COUNT/0/  IN/5/, OUT/6/, NCARDS/100/
C. . . . . .ВВОД ДАННЫХ . . . . . . . . . . . . . . . . . . . .
      DO 10 I=1,NCARDS
         READ (IN,100,END=15) X(I)
   10 CONTINUE
      GO TO 20
C. . . . . .ЕСЛИ МЕНЬШЕ 100 ЗНАЧЕНИЙ . . . . . . . . . . . . .
C. . . . . .ПРИНЯТЬ ПОСЛЕДНЕЕ В КАЧЕСТВЕ КРИТЕРИЯ . . . . .
   15 NCARDS = I-1
      IF (NCARDS .GT. 1) GO TO 20
         WRITE (OUT,102) NCARDS
         STOP
C. . . . . .НАЙТИ ПОСЛЕДНЕЕ ВВЕДЕННОЕ ЗНАЧЕНИЕ. . . . . . .
   20 FINAL = X(NCARDS)
      NCM1 = NCARDS-1
      DO 30 I = 1,NCM1
         IF (X(I) .GT. FINAL) COUNT = COUNT+1
   30 CONTINUE
C. . . . . .ВЫВЕСТИ ОКОНЧАТЕЛЬНЫЕ РЕЗУЛЬТАТЫ. . . . . . .
      WRITE (OUT,101) NCARDS, COUNT, FINAL
      STOP
  100 FORMAT (I5)
  101 FORMAT (5X, 'OF THE ', I3, ' NUMBERS INPUT,'
     1  I3, ' VALUES EXCEEDED THE FINAL VALUE OF ', I6)
  102 FORMAT (' ***ERROR: ONLY ', I3, 'NUMBERS INPUT...ABORT***')
      END
```

Рис.0.8. Использование индекса цикла после выхода из цикла.
(В инструкции **101**: ''ИЗ', 'ВВЕДЕННЫХ ЧИСЕЛ', 'ПРЕВЫШАЮТ ПОСЛЕДНЕЕ ЗНАЧЕНИЕ'; в инсгрукции **102**; '***ОШИБКА: ТОЛЬКО', 'ВВЕДЕННЫХ ЧИСЕЛ ... ОСТАНОВ ***')

инструкция IF, расположенная за инструкцией с меткой 15, обеспечивает, чтобы цикл DO не выполнялся второй раз, если было введено недостаточное количество чисел. Без этой инструкции во время второго прохода цикла DO программа выполнялась бы неверно из-за конечного значения NCARDS, равного нулю.

0.7. Подпрограммы

0.7.1. Введение

Подпрограммы служат важным средством для написания на Фортране сложных программ. При правильном использовании подпрограмм существенно легче писать, отлаживать, документировать программы. Такие программы удобнее в эксплуатации.

О роли подпрограмм в модульных структурах мы поговорим в гл. 1. В данном разделе мы рассмотрим только технику построения самих подпрограмм.

Подпрограмма — это независимый программный блок. Подпрограммы во многом схожи с главными программами, которые уже были нами описаны. Подпрограммы и главные программы образуют *программные единицы* или *программные модули*. Мы говорим: подпрограмма *вызывается* из другого программного модуля. Во время вызова подпрограммы управление передается из вызывающей программы (или подпрограммы) в вызываемую подпрограмму. После этого начинается выполнение подпрограммы, состоящей из обычных инструкций Фортрана, таких, как инструкции присваивания, передачи управления (IF, GO TO, DO) и ввода/вывода. Подпрограммы, могут также вызывать другие подпрограммы, декларировать тип данных, резервировать память и задавать начальные значения с помощью инструкций REAL, INTEGER, LOGICAL, DIMENSION, IMPLICIT и DATA. В какой-то момент времени вызванная подпрограмма *передает управление обратно* в вызывающий программный модуль. Это означает, что управление будет передано на инструкцию, непосредственно следующую за той инструкцией, из которой произошел вызов подпрограммы. (В гл. 2 будет показано, что управление может быть передано и другим инструкциям вызывающей программы.)

Все программные модули компилируются независимо; это означает, что взаимосвязи между метками инструкций или именами переменных в двух программных модулях нет. Таким образом, имя K, например, или номер инструкции 10 можно использовать в нескольких программных модулях без всяких опасений.

Передача данных (параметров) между программными модулями происходит не по имени, так как нет взаимосвязи между двумя, даже одинаковыми именами в различных программных модулях. Существует три способа обмена информации между вызывающей и вызываемой программами. Первый способ применяется в *подпрограммах-функциях*, которые возвращают одно-единственное арифметическое или логическое значение. Этот способ обеспечивает только одностороннюю связь: из подпрограммы-функции к вызывающей ее программе. Двухсторонняя связь возможна с помощью *списка параметров*, т. е. ряда значений и переменных, которые вызывающая программа передает подпрограмме в момент вызова. Подпрограмма может использовать эти значения для вычислений и присваивать результаты переданным ей переменным. В момент возврата, вызывающий программный модуль получает доступ к результатам, которые подпрограмма присвоила этим переменным. Третий способ позволяет разделять одни и те же данные между двумя программными модулями. Он основан

на применении инструкции COMMON. В этом случае переменные, которые были описаны в инструкции COMMON, хранятся в области памяти, доступной любому программному модулю. В данном разделе мы проанализируем только два первых способа связи между программными модулями, т. е. передачу результата как значения функции и обмен данными при использовании списков параметров; инструкция COMMON будет рассмотрена в разд. 2.6.

0.7.2. Списки параметров

Как было сказано выше, список параметров — это последовательность значений и переменных, передаваемых между программными модулями. Список параметров состоит из одного или нескольких элементов, разделенных запятыми; список заключен в круглые скобки. Элементами списка параметров могут быть константы, простые переменные, выражения, имена массивов и элементы массивов. Список параметров в вызывающей программе называется списком *фактических параметров*, поскольку при обращении к подпрограмме ей передаются фактические значения и переменные.

Подпрограмма должна содержать аналогический список, называемый списком *формальных параметров*. Этот список состоит только из простых переменных и имен массивов. Поскольку имена в различных программных модулях не связаны между собой, между списками формальных и фактических параметров устанавливается позиционное соответствие: первый фактический параметр связывается с первым формальным параметром и т. д. (Нередко для наглядности программы и удобства работы с ней в обоих списках параметров записывают одинаковые имена. Следует, однако, помнить, что связь между формальными и фактическими параметрами определяется их позициями в списках, а не их именами.) Формальные переменные являются лишь «представлением» тех фактических переменных и значений, которые заменят их при обращении к подпрограмме. Каждое изменение формального параметра, например, инструкцией присваивания или инструкцией READ имеет тот же эффект, что и изменение соответствующего фактического параметра.

В связи с тем что формальный параметр является только представлением фактического параметра, они должны быть совместимыми. Мы называем это *согласованием параметров*. Формальные и фактические параметры должны быть согласованы в следующих аспектах.

По типу: каждый фактический параметр должен быть того же типа, что и соответствующий ему формальный параметр.

По роду: каждому одиночному элементу данных (константе, простой переменной или выражению) в списке формальных параметров должна соответствовать простая переменная. Элементу массива в списке фактических параметров обычно соответствует простая переменная в списке формальных параметров; но ему может также соответствовать имя массива в списке формальных параметров. В этом последнем случае с передаваемым элементом массива и всеми его последующими элементами будут сопоставляться первый, второй и т. д. элементы массива в списке формальных параметров.

По размерности: поскольку при обращении к элементам массива применяется формула доступа, массив в списке фактических параметров должен иметь ту же размерность, что и массив в списке формальных параметров. Исключение представляет лишь одномерный массив, или вектор, в котором размерность «формального» массива может быть меньше размерности «фактического» массива. Фактические размеры массива могут быть также переданы в качестве параметров.

По количеству: количество элементов в списках фактических и формальных параметров должно совпадать.

Существует две разновидности подпрограмм: подпрограммы типа FUNCTION (далее функции) и подпрограммы типа SUBROUTINE (далее подпрограммы). Они отличаются друг от друга как способом вызова, так и способом возвращения результатов.

Функции обычно возвращают только одно арифметическое или логическое значение. Для вызова функции используется ее имя, за которым в скобках указывается список формальных параметров. Функция выступает в качестве терма в арифметическом или логическом выражении. Результат вычисления функции подставляется как значение этого терма и участвует в дальнейших вычислениях совместно с другими термами этого выражения. Вызов функции (или обращение к функции) имеет следующую форму:

имя (спис-парам)

где *имя* — имя функции;

спис-парам — список простых переменных, элементов массивов, констант или выражений;

все вместе — терм арифметического или логического выражения.

Пример:

```
REAL P, BIG
P=5.0
GROSS=BIG (P, —1.75)
```

Действие: Переменной GROSS присваивается значение, вычисленное функцией BIG с фактическими параметрами P и —1.75.

0.7.3. Встроенные функции

Ряд функций широко используется в решении многих задач. К ним относится вычисление квадратного корня, синуса и косинуса, логарифма, абсолютного значения и т. д. Программы для вычисления этих функций входят в качестве составной части в большинство компиляторов Фортрана, поэтому такие функции называют *встроенными*. Мы включили в приложение 1 таблицу наиболее употребительных встроенных функций. Обратите внимание на указанные в таблице типы параметров. При обращении к каждой функции следите за тем, чтобы параметры были требуемых типов. На этом мы заканчиваем рассмотрение математических функций, поскольку их применение довольно очевидно. Остановимся, однако, на некоторых функциях иного рода.

Прежде всего обратимся к функциям FLOAT и IFIX. Эти функции позволяют изменить тип выражения так, чтобы он отличался от типа его аргументов. Функция FLOAT возвращает вещественное число, эквивалентное значению ее целого параметра. Функция IFIX возвращает целое число, эквивалентное значению ее вещественного параметра с усеченной дробной частью. Эти функции оказываются весьма полезными, когда необходимо согласовать типы параметров при вызове подпрограммы. Предположим требуется вычислить квадратный корень из целого K. Таблица, приведенная в приложении 1, указывает, что функция SQRT вычисляет квадратные корни, но ее аргумент должен быть вещественного типа. Поэтому запись SQRT(K) была бы ошибочной. Одно из решений проблемы — задать вещественную переменную, скажем AK, и присвоить ей значение K, т. е. AK=K. Однако при этом вводится лишняя переменная и программа становится менее понятной. Можно принять другое решение — применить функцию FLOAT для получения вещественного числа, эквивалентного K. Запись SQRT (FLOAT(K)) позволяет вычислить квадратный корень из данного целого числа K. Программа становится короче и не вводит в заблуждение читателя лишними переменными, которые используются только один раз.

Функции MAX и MIN принимают переменное число параметров и возвращают соответственно наибольшее или наименьшее значение полученных параметров. Предположим, что нужно выполнить цикл как минимум один раз, но не более 20 раз, и если значение функции находится в диапазоне между 1 и 20, то цикл должен выполняться (K+1)/2 раз. Это может быть достигнуто с помощью следующих инструкций:

```
NTIMES=MAXO(1, MINO((K+1)/2,20))
DO 55 KOUNT=1, NTIMES
```

Приведенные в приложении 1 четыре функции MAX отличаются друг от друга типом параметров и конечных результатов.

AMAX0 и AMAX1 в качестве результата возвращают вещественные значения, а MAX0 и MAX1 — целые значения. Функции AMAX0 и MAX0 принимают только целые параметры, а AMAX1 и MAX1 — вещественные. Функции MIN определяются аналогично. Все параметры функций MAX и MIN должны быть заданы явно: они могут быть простыми переменными, константами, выражениями или элементами массива. Имена массивов не допускаются. По этой причине, даже если TABLE — это массив целых из 20 элементов, то запись MAX0 (TABLE) *не* может быть использована для определения наибольшего элемента этого массива.

0.7.4. Подпрограммы-функции, определенные пользователем

Подпрограммы, определенные пользователем, состоят из тех же инструкций, что и главные программы: они включают декларации, инструкции присваивания, инструкции READ, WRITE, FORMAT, IF, GO TO, DO, STOP. Подобно главной программе, каждая подпрограмма, определенная пользователем, должна оканчиваться инструкцией END, отмечающей физический конец программного модуля. Дополнительная инструкция подпрограммы, RETURN, осуществляет возврат управления в точку вызова. Эта инструкция имеет следующую форму:

[*метка*] RETURN

где *метка* — (необязательный) номер инструкции.

Пример:

```
20 RETURN
```

Действие: Возобновляется выполнение программы в той точке, из которой была вызвана данная подпрограмма. В момент возврата значения всех формальных параметров становятся значениями соответствующих фактических параметров.

Определение функции начинается с инструкции FUNCTION, в которой указываются тип результата, возвращаемого функцией, имя функции и имена формальных параметров. Она имеет следующий вид:

[*тип*] FUNCTION *имя (спис-парам)*

где *тип* — явно указанный тип результата функции. *По умолчанию*, т. е. если тип функции не указан, он определяется *именем* в соответствии со стандартным соглашением о связи между типом переменной и первой буквой ее имени;

имя — имя функции; оно же используется для возвращения результата. Имя функции образуется по тем же правилам, что и имя переменной;
спис-парам — список простых переменных или имен массивов, содержащий формальные параметры этой функции.

Пример:

```
REAL FUNCTION BIG(X, Y)
```

Действие: Декларируется функция вещественного типа BIG; указывается, что это начало определения функции; указываются два формальных параметра: X и Y.

Теперь рассмотрим несколько примеров.

Пример:

```
REAL FUNCTION BIG (X,Y)
REAL X,Y
BIG = X
IF (X .LT. Y) BIG = Y
RETURN
END
```

Действие: Из двух вещественных значений X и Y функция BIG возвращает в качестве результата наибольшее.

Примечания:

1. Значение должно быть присвоено имени функции до выполнения инструкции RETURN, так как результат подпрограммы-функции возвращается как значение имени функции.
2. Внутри подпрограммы-функции имя функции может быть использовано подобно любой неиндексированной переменной.
3. Функция не может вызывать сама себя ни прямо (A не может вызывать A), ни косвенно (если A вызывает B, то ни B, ни какая-либо другая подпрограмма, вызываемая B, не могут вызывать A).
4. Функция не должна изменять значения формальных параметров.

Имя подпрограммы-функции используется в данном примере двояким образом. Во время выполнения подпрограммы имя — это обычная переменная, и она может участвовать в вычислениях. В момент выполнения инструкции RETURN текущее значение имени функции становится результатом, возвращаемым подпрограммой-функцией. Возвращение результата через имя функции — обычно единственное средство, с помощью которого функция связывается с программой, которая ее вызывает.

Рассмотрим пример обращения к функции BIG:

```
REAL A,B,C,D
READ (-,-) A,B,C
D = BIG (A,B)
D = BIG (D,C)
WRITE (-,-) D
```

При первом вызове BIG текущие значения A и B замещают значения формальных параметров X и Y соответственно. Вначале BIG присваивается значение X(=A). Однако если Y (=B) больше, то значение BIG замещается на Y. Инструкция RETURN пересылает текущее значение BIG обратно в правую часть первой инструкции присваивания. При втором вызове BIG предыдущий результат D будет передан обратно в подпрограмму-функцию BIG в качестве значения параметра X, а значение C — в качестве значения параметра Y. Снова выполняются те же действия и BIG возвращает наибольшее из двух значений, D или C. Выражение BIG (A, B) можно использовать в качестве одного из параметров функции BIG. Инструкция

D=BIG(BIG(A, B), C)

дает такой же результат, как и две приведенные выше инструкции присваивания.

Прежде чем перейти к подпрограммам типа SUBROUTINE, рассмотрим еще один пример с использованием подпрограммы-функции. В этом примере содержится несколько вызовов функции AVG, которая вычисляет среднее значение первых N элементов вектора из 100 вещественных чисел.

```
      REAL X(100), Y(100)
      REAL AVG, RES
        :
    1 RES = AVG(X,10)
        :
    2 IF (AVG(Y,J) .LE. 0) GO TO 5
        :
    3 X(I) = X(I) - AVG(Y, 100-I)
        :
        :
      STOP
      END
```

```
      REAL FUNCTION AVG (X,N)
      REAL X(100)
      INTEGER N
      AVG = 0.
      IF (N.LT.1 .OR. N.GT.100) RETURN
      DO 1 I = 1, N
         AVG = AVG + X(I)
    1 CONTINUE
      AVG = AVG/N
      RETURN
      END
```

Функция AVG в этой программе вызывается для выполнения три раза (если инструкции с номерами 1, 2 и 3 выполняются по одному разу). При первом обращении к подпрограмме-функции вычисляется среднее значение первых 10 элементов вектора X. При втором обращении вектор Y и переменная J передаются подпрограмме-функции в качестве параметров, а результат представляет собой среднее значение J элементов, от Y(1) по Y(J). При третьем обращении в качестве передаваемого параметра вы-

ступает выражение: если I имеет значение 32, то результат вычисления функции — это среднее значение элементов от Y (1) по Y (68).

Когда в результате какого-то вычисления необходимо вернуть одно результирующее значение, удобно воспользоваться подпрограммой-функцией. Однако для организации более сложных связей необходимы подпрограммы другого типа, такие, как *SUBROUTINE*. Перейдем к их рассмотрению.

0.7.5. Подпрограммы типа SUBROUTINE

Подпрограммы такого типа вызываются с помощью специальной инструкции CALL. Эта инструкция имеет следующую форму:

[*метка*] CALL *имя* [(*спис-парам*)]

где *метка* — (необязательный) номер инструкции;
имя — имя подпрограммы;
спис-парам — список простых переменных, констант, выражений, элементов и имен массивов.

Пример:

```
REAL ALARGE, P
DATA P/5.0/
CALL MAX (P, 4.7, ALARGE)
```

Действие: Вызывается подпрограмма MAX и ей передаются фактические параметры P, 4.7, ALARGE.

Подпрограммы, написанные пользователем, начинаются с инструкции SUBROUTINE:

SUBROUTINE *имя* [(*спис-парам*)]

где *имя* — имя, которое идентифицирует подпрограмму; оно строится по тем же правилам, что и имя переменной;
спис-парам — список из одной или нескольких простых переменных или имен массивов, разделенных запятыми.

Пример:

```
SUBROUTINE ORDER (X, Y)
```

Действие: Задается начало подпрограммы с именем ORDER и двумя формальными параметрами X и Y.

Тело подпрограммы типа SUBROUTINE строится подобно телу функции; в подпрограмме используются те же декларации, присваивания, переходы и инструкции ввода/вывода. Инструкция RETURN, как и в функции, передает управление обратно в точку, из которой была вызвана подпрограмма. Однако в отли-

чие от имени функции имя подпрограммы типа SUBROUTINE *нельзя* использовать в качестве переменной. Оно служит только для идентификации подпрограммы. Другое отличие состоит в том, что в подпрограмме разрешено изменять значения формальных параметров; и именно таким путем подпрограммы возвращают результаты вызывающему программному модулю. Приведем примеры подпрограмм.

Пример:

```
SUBROUTINE ORDER (X,Y)
REAL X, Y, LARGER
IF (X .GT. Y) RETURN
LARGER = Y
Y = X
X = LARGER
RETURN
END
```

Действие: Из двух заданных вещественных значений X и Y подпрограмма ORDER наибольшее значение присваивает X, а наименьшее Y.

Примечания:

1. Значения переменных в списке параметров могут быть изменены в ходе выполнения подпрограммы.
2. Инструкция RETURN может быть расположена в любом месте исполняемой части подпрограммы.
3. Имя подпрограммы не является переменной; оно только идентифицирует подпрограмму. Имя не имеет типа.
4. Никакая другая подпрограмма или переменная в данной подпрограмме, а также в вызывающем программном модуле не могут иметь имени, идентичного с именем данной подпрограммы.
5. Подпрограмма не может вызывать сама себя ни прямо (A не может вызывать A), ни косвенно (если A вызывает B, то ни B, ни другая подпрограмма, которую вызывает B, не могут вызывать A).

Механизмы возврата результатов в подпрограммах и функциях различаются. В то время как функция возвращает единственный результат как значение имени функции, подпрограмма передает результаты как значения своих параметров. В момент выполнения инструкции RETURN конечные значения всех формальных параметров присваиваются соответствующим фактическим параметрам. Следует, однако, иметь в виду, что, если изменяется значение формального параметра, представляющего собой константу, могут возникнуть неожиданные результаты или даже ошибки. В разд. 2.2 мы рассмотрим различные механизмы

передачи параметров; пока же ограничимся простым советом: можно изменять значение только такого формального параметра, которому в списке фактических параметров соответствует простая переменная, элемент или имя массива.

Так как подпрограмма возвращает результаты вычислений через список параметров, она может передавать несколько значений. Каждой простой переменной в списке параметров подпрограмма может присваивать значение, которое в момент возврата передается обратно в вызывающую программу. Если в списке формальных параметров указан массив, то в результате выполнения подпрограммы можно передать значение каждого элемента массива. Таким образом, число результатов, возвращаемых подпрограммой SUBROUTINE, ограничено только числом формальных параметров и размерностью массивов, выступающих в роли формальных параметров.

С другой стороны, в подпрограмме вообще может отсутствовать список параметров. Рассмотрим подпрограмму, которая предназначена для повторной печати заголовка отчета. Вызов подпрограммы происходит каждый раз в ходе печати, когда начинается новая страница. При этом подпрограмме не требуется передавать никакой информации, и нет никаких результатов, которые следовало бы передавать из подпрограммы обратно в вызывающую программу. В такой подпрограмме не нужны никакие параметры. Инструкции SUBROUTINE и CALL для такой подпрограммы не содержат списка параметров и круглых скобок.

Приведем пример еще одной подпрограммы. Подпрограмма INPUT считывает N значений, располагает их в массиве и присваивает переменной ERROR значение TRUE, если данных не хватает.

```
      REAL X(30), Y(30), Z(40)
      LOGICAL EOF
      INTEGER M
      M = 30
      CALL INPUT (X, 10, EOF)
      IF (EOF) GO TO 99
        :
      CALL INPUT (Y, M, EOF)
        :
      CALL INPUT (Z, M+10, EOF)
        :
   99 WRITE (6,199)
  199 FORMAT (' OUT OF DATA')
      STOP
        :
```

```
      SUBROUTINE INPUT (X, N, ERROR)
      REAL X(N)
      LOGICAL ERROR
      ERROR = .FALSE.
      IF (N.LE.0) RETURN
      READ (5,4,END=1) (X(I),I=1,N)
      RETURN
    1 ERROR = .TRUE.
      RETURN
    4 FORMAT (16I5)
      END
```

(Сообщение в инструкции 199: 'не хватает данных')

Выполнение подпрограммы при первом ее вызове происходит следующим образом:

1. В момент вызова значения элементов вектора X, константа 10 и переменная EOF передаются в подпрограмму INPUT. В под-

программе им соответствуют формальные параметры X, N и ERROR.

2. В подпрограмме INPUT выполнение операций продолжается до тех пор, пока не будет считано N значений или не будут введены все перфокарты.

3. При выполнении инструкции RETURN текущие значения элементов вектора X, переменных N и ERROR присваиваются фактическим параметрам, в данном случае X, 10 и EOF.

4. В вызывающей программе выполнение возобновляется с инструкции, непосредственно следующей за инструкцией CALL.

Заметим, что в данном примере размерность массива передается в качестве параметра и что формальный параметр X описан в декларации REAL как массив. Массивам, передаваемым в качестве параметров, не отводится область памяти в подпрограмме. Декларация REAL необходима лишь для того, чтобы указать, что X — массив, а размеры массива используются в формуле доступа для ссылок на элементы массива. Массив может иметь переменный индекс в декларации только в том случае, если *и имя* массива, *и числа*, *ограничивающее диапазоны* изменения его индексов, являются формальными параметрами. Кроме того, эти числа (размеры) должны быть целыми. В таких случаях говорят об использовании *регулируемых размеров*.

Как показано в приведенном примере при первом вызове подпрограммы, регулируемый размер массива может быть меньше, чем фактический размер массива, однако он не может быть больше фактического размера. Для многомерных массивов передаваемые размеры должны быть точно согласованы с истинными размерами массивов, так как передаваемые размеры используются для расчетов по формуле доступа.

На этом мы заканчиваем описание основных инструкций Фортрана. Глава завершается примером, в котором использованы многие из рассмотренных нами положений. Когда вы приступите к разбору примера, вероятно, вам будет нужно вернуться к определению тех или иных инструкций для того, чтобы уточнить, как они работают или почему они так записаны.

0.8. Пример

В этом разделе мы рассмотрим пример, в котором покажем применение инструкций, описанных в данной главе. Пусть фирма INFO — организация, специализирующаяся на поставках товаров, заказанных по почте. У фирмы есть список фамилий и адресов потенциальных заказчиков; этот список отсортирован по шифрам клиентов. Фирма очень заинтересована в получении новых имен и адресов, которые могут быть добавлены к основному списку. В связи с этим фирма покупает списки имен и адресов

у специальных агентов. Получая списки от агентов, фирма оплачивала их услуги. Позже однако выяснилось, что многие «новые» клиенты в действительности дублируют тех, что уже были в списке. Поэтому необходимо иметь такую программу, которая обнаруживает одинаковые имена, обновляет основной список и начисляет агенту $ 25 за каждого действительно нового клиента.

Для упрощения программы мы будем полагать, что ввод организован с перфокарт, основной файл состоит не более чем из 1000 пятиразрядных шифров клиентов, каждый список добавлений упорядочен, содержит не более 100 записей (шифров) и заканчивается перфокартой, на которой пробито 99999 [1]. Общая блок-схема решения данной задачи приведена на рис. 0.9. После чтения основного файла и вплоть до обнаружения «конца файла» (EOF) для колоды перфокарт с новыми данными выполняется следующий цикл:

1. Прочитать очередной файл добавлений.

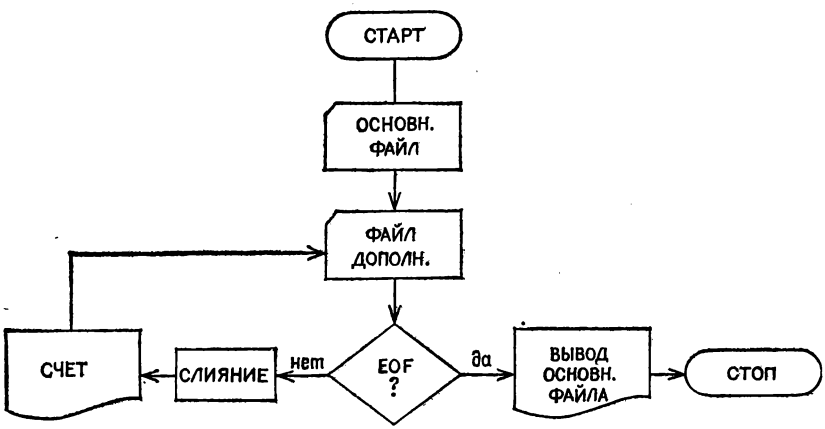


Рис.0.9. Общая блок-схема решения задачи.

2. Выполнить слияние с основным файлом, устранив дублирование.
3. Вывести на печать счет к оплате за новый список.

Ниже приведен пример, показывающий, как должна работать та часть программы, которая обеспечивает слияние. В результате слияния основного списка (MASTER) со списком добавлений (UPDATE) получается новый файл (MERGED FILE).

---

[1]) Из программы (рис. 0.11) непосредственно следует, что список добавлений начинается с номера, который, по-видимому, является шифром агента.— *Прим. ред.*

MASTER	UPDATE	MERGED FILE	
1	2	1	
3	3	2	<== НОВЫЙ КЛИЕНТ
8	5	3	
10	8	5	<== НОВЫЙ КЛИЕНТ
		8	
		10	

Можно, конечно, выполнить слияние таким образом, чтобы результирующий файл после слияния был бы тем же самым массивом, который первоначально содержал основной список. Но для этого потребовалось бы дважды просматривать основной файл. Поэтому мы будем хранить оба основных списка в массиве, состоящем из двух столбцов, один из которых представляет текущий основной список, а другой — результат слияния.

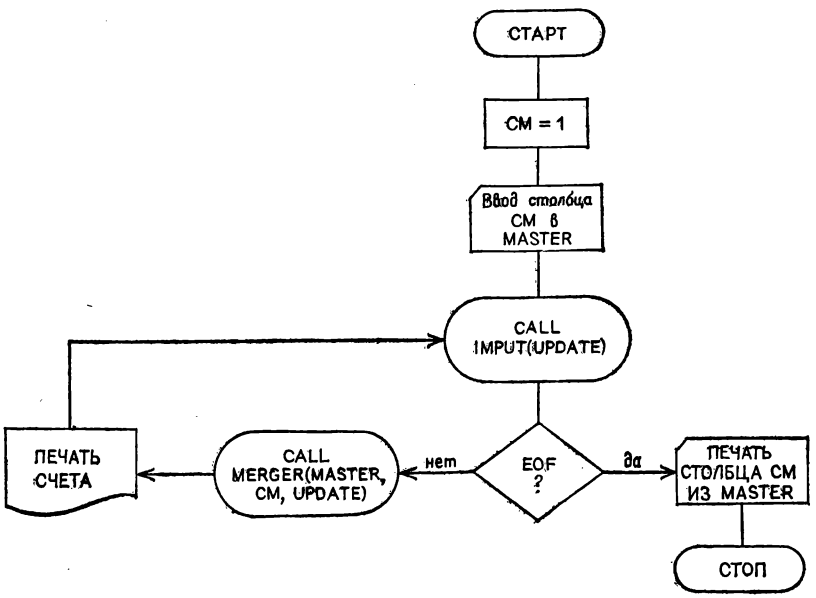


Рис.0.10. Блок-схема главной программы.

Мы должны иметь возможность выполнять попеременные слияния, если на входе будет более одного файла добавлений: первоначально будет произведено слияние с первым столбцом, затем со вторым, затем снова с первым и т. д. Вместо того чтобы писать две одинаковые программы для выполнения того же самого слияния, мы создадим подпрограмму MERGER, которая будет работать в обоих случаях. Аналогично воспользуемся подпрограм-

```
C==================================================
C=ПРОГРАММА СЛИЯНИЯ И НАЧИСЛЕНИЯ СЧЕТА=            =
C==================================================
C=                                                =
C=  ПРОГРАММА ОБНОВЛЯЕТ ОСНОВНОЙ ФАЙЛ ИДЕНТИФИКАЦИОННЫХ=
C=  НОМЕРОВ КЛИЕНТОВ С ПРОИЗВОЛЬНЫМ ЧИСЛОМ ВХОДНЫХ=
C=  ФАЙЛОВ. ВЫПОЛНЯЕТСЯ СЛИЯНИЕ, ДУБЛИРУЮЩИЕСЯ НОМЕРА=
C=  ОПУСКАЮТСЯ. ПРИ КАЖДОМ ДОБАВЛЕНИИ НОВОГО НОМЕРА=
C=  К ОСНОВНОМУ ФАЙЛУ НАЧИСЛЯЕТСЯ.0.25 НА СЧЕТ АГЕНТА. ПОСЛЕ=
C=  КАЖДОГО ОБНОВЛЕНИЯ ПЕЧАТАЕТСЯ СУММАРНАЯ ЗАДОЛЖЕН-=
C=  НОСТЬ. ПОСЛЕ ЗАВЕРШЕНИЯ ВСЕХ ОБНОВЛЕНИЙ РАСПЕЧАТЫВАЕТСЯ=
C=  НОВЫЙ ОСНОВНОЙ ФАЙЛ.                          =
C=                                                =
C=   ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                          =
C=     UPDATE — ФАЙЛ ДОБАВЛЕНИЙ НА ПЕРФОКАРТАХ;   =
C=     MASTER — ОСНОВНОЙ ФАЙЛ, ДВА СТОЛБЦА ОСНОВНОГО ФАЙЛА =
C=              ИСПОЛЬЗУЮТСЯ ПОПЕРЕМЕННО;        =
C=     OMCOL  — ОПРЕДЕЛЯЕТ, КАКОЙ СТОЛБЕЦ ФАЙЛА MASTER=
C=              СОДЕРЖИТ СТАРЫЕ ДАННЫЕ;           =
C=     UPSIZ  — ЧИСЛО ИДЕНТИФИКАЦИОННЫХ НОМЕРОВ В ПОСЛЕД-=
C=              НЕМ ФАЙЛЕ ДОБАВЛЕНИЙ;             =
C=     UPNUM  — НОМЕР ФАЙЛА UPDATE;               =
C=     OMSIZ  — ЧИСЛО ИД.НОМЕРОВ В ПОСЛЕДНЕМ MASTER;=
C=     COST   — СТОИМОСТЬ ПОСЛЕДНЕГО ОБНОВЛЕНИЯ.   =
C=                                                =
C=  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                       =
C=    INPUT  — ПОДПРОГРАММА ВВОДА ВСЕХ ФАЙЛОВ UPDATE;=
C=   MERGER  — ПОДПРОГРАММА СЛИЯНИЯ ПОСЛЕДНЕГО UPDATE =
C=              С ФАЙЛОМ MASTER.                  =
C=                                                =
C=ПРОГРАММИСТ: — — —          ДАТА: — — —         =
C==================================================

C=================================================================
C
      INTEGER UPDATE(100), MASTER(1000,2), UPSIZ, UPNUM, OMSIZ, OMCOL
      INTEGER RDR, PRT
      REAL COST
      DATA OMCOL/1/, RDR/5/, PRT/6/
C------------ ВВЕСТИ И РАСПЕЧАТАТЬ ФАЙЛ MASTER
      READ (RDR,100,END=99) OMSIZ, (MASTER(I,OMCOL), I=1,OMSIZ)
  100 FORMAT (I5 / (16I5) )
      WRITE (PRT,110) (MASTER(I,OMCOL), I=1,OMSIZ)
  110 FORMAT ('1', 20X, 'CURRENT MASTER FILE:' // 8(5X, I5) )
C------------ ВЫПОЛНИТЬ ЕЩЕ ОДНО СЛИЯНИЕ?
   10 CALL INPUT (UPDATE, UPSIZ, UPNUM)
      IF (UPSIZ .EQ. 0) GO TO 20
C------------ РАСПЕЧ. UPDATE, СЛИТЬ И ВЫВЕСТИ ЗАДОЛЖЕННОСТЬ
         WRITE (PRT,120) UPNUM, (UPDATE(I), I=1,UPSIZ)
  120    FORMAT ('0', 20X, 'UPDATE FILE NUMBER', I6, ':' // 8(5X, I5) )
         CALL MERGER (UPDATE, UPSIZ, MASTER, OMSIZ, OMCOL)
         COST = .25*UPSIZ
         WRITE (PRT,130) UPNUM, COST, UPSIZ
  130    FORMAT ('0', 2X, 'FILE NUMBER', I6, ' TO BE PAID $', F6.2,
     X      ' FOR', I5, ' NEW ADDRESSES')
         GO TO 10
C------------ РАБОТА ОКОНЧЕНА: ВЫВОД MASTER И ОСТАНОВ
   20 WRITE (PRT,110) (MASTER(I,OMCOL), I=1,OMSIZ)
      WRITE (PRT,140)
  140 FORMAT ('0', '*** JOB COMPLETED ***')
      STOP
C------------ ОШИБКА ВО ВХОДНОМ ФАЙЛЕ: ПРЕКРАТИТЬ РАБОТУ
   99 WRITE (PRT,150)
```

Рис. 0.11. Программа слияния файлов и подготовки счета для оплаты. (Текст в инструкции **110**: «Старый файл MASTER», **120**; «Номер файла UPDATE», **130**: «Номер файла», «Выплатить», «Новые адреса», **140**: «Работа закончена», **150**: «Ошибка при вводе MASTER, останов», **110**: «Слишком много добавлений»; **100**: «Превышен размер MASTER, останов в MERGER».)

```
  150 FORMAT (' *** MASTER FILE INPUT ERROR: HALTING IN MAIN ***')
      STOP
      END

      SUBROUTINE INPUT (FILE, N, FNUM)
C=================================================================
C= ЕСЛИ ПЕРЕД ЭТИМ НЕ ВСТРЕТИЛСЯ КОНЕЦ ФАЙЛА,ВВЕСТИ СЛЕДУЮЩИЙ ФАЙЛ =
C= ДОБАВЛЕНИЙ                                                     =
C= ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                           =
C=     FILE   — ПАРАМЕТР, ОБЛАСТИ ДЛЯ НОВОГО ФАЙЛА UPDATE         =
C=     N      — ПАРАМЕТР, РАЗМЕР НОВОГО ФАЙЛА UPDATE              =
C=     FNUM   — ПАРАМЕТР, НОМЕР НОВОГО ФАЙЛА                       =
C=     EOF    — TRUE. ЕСЛИ ВСТРЕТИЛСЯ КОНЕЦ ФАЙЛА                  =
C=     MARKER — ПРИЗНАК КОНЦА ДОБАВЛЕНИЙ: ЗДЕСЬ 99999              =
C=================================================================
      INTEGER FILE(100), N, FNUM
      INTEGER NEWVAL, MARKER, RDR, PRT
      LOGICAL EOF
      DATA MARKER/99999/, EOF/.FALSE./, RDR/5/. PRT/6/
C------------ЕСЛИ УЖЕ БЫЛ КОНЕЦ ФАЙЛА,ТО ПРЕКРАТИТЬ РАБОТУ
      I = 1
      IF (.NOT. EOF) GO TO 10
         N = 0
         RETURN
C------------ВВЕСТИ НОМЕР ФАЙЛА И ВСЕ ДОБАВЛЕНИЯ, ВПЛОТЬ ДО MARKER
   10 READ (RDR,100,END=30) FNUM
  100 FORMAT (I5)
      DO 20 I = 1, 100
         READ (RDR,100,END=30) FILE(I)
         IF (FILE(I) .EQ. MARKER) GO TO 40
   20 CONTINUE
C------------ДОЛЖЕН БЫТЬ ИЛИ МАРКЕР ИЛИ КОНЕЦ ФАЙЛА
      I = 101
      READ (RDR,100,END=30) NEWVAL
      IF (NEWVAL .EQ. MARKER) GO TO 40
C------------СЛИШКОМ МНОГО (БОЛЕЕ 100) ДОБАВЛЕНИЙ
      WRITE (PRT,110) FNUM
  110 FORMAT (' *** TOO MANY UPDATES IN FILE', I6, ' ***')
      STOP
C------------КОНЕЦ ФАЙЛА: ЗАПОМНИТЬ ДЛЯ СЛЕДУЮЩЕГО ВЫЗОВА
   30 EOF = .TRUE.
   40 N = I-1
      RETURN
      END
```

```
      SUBROUTINE MERGER (UPDATE. UPSIZ, MASTER, OMSIZ, OMCOL)
C=================================================================
C= СЛИЯНИЕ СПИСКА ДОБАВЛЕНИЙ СО СТОЛБЦОМ OMCOL МАССИВА=
C= MASTER; ЗАНЕСЕНИЕ РЕЗУЛЬТАТА В ДРУГОЙ СТОЛБЕЦ MASTER =
C=                                                                =
C= ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                           =
C=    UPDATE — СПИСОК, ПОДЛЕЖАЩИЙ СЛИЯНИЮ С MASTER;               =
C=    UPSIZ  — РАЗМЕР ФАЙЛА UPDATE;                               =
C=    MASTER — СТОЛБЕЦ OMCOL — ЭТО MASTER; В ДРУГОЙ СТОЛБЕЦ=
C=             ПОМЕЩАЕТСЯ НОВЫЙ MASTER;                           =
C=    OMSIZ  — ЧИСЛО ЭЛЕМЕНТОВ В СТАРОМ MASTER;                   =
C=    OMCOL  — СТОЛБЕЦ 'MASTER', СОДЕРЖАЩИЙ СТАРЫЙ MASTER;        =
C=    NMCOL  — СТОЛБЕЦ 'MASTER' ДЛЯ НОВОГО MASTER;                =
C=    UPPTR  — СЧЕТЧИК ДЛЯ СПИСКА UPDATE;                         =
C=    OMPTR  — СЧЕТЧИК ДЛЯ СТАРОГО MASTER;                        =
C=    NMPTR  — СЧЕТЧИК ДЛЯ НОВОГО MASTER;                         =
C=    MSIZ   — РАЗМЕР ФАЙЛА MASTER (ЗДЕСЬ — 1000);                =
C=================================================================
```

Продолжение Рис. 0.11.

```
      INTEGER UPDATE(100), UPSIZ, MASTER(1000,2), OMSIZ, OMCOL, NMCOL
      INTEGER UPPTR, OMPTR, NMPTR, PRT
      DATA PRT/6/
C------------- NMCOL ДОЛЖЕН УКАЗЫВАТЬ НА СТОЛБЕЦ НОВОГО MASTER
      NMCOL = 1
      IF (OMCOL .EQ. 1) NMCOL=2
C------------- НАЧИНАТЬ С ВЕРШИНЫ КАЖДОГО СПИСКА
      UPPTR = 1
      OMPTR = 1
      NMPTR = 1
C------------- ОПРЕДЕЛИТЬ, ИЗ КАКОГО СПИСКА ВЗЯТЬ СЛЕДУЮЩИЙ ЭЛЕМЕНТ
   10 IF (UPPTR.GT.UPSIZ .AND. OMPTR.GT.OMSIZ) GO TO 50
        IF (UPPTR .GT. UPSIZ) GO TO 40
        IF (OMPTR .GT. OMSIZ) GO TO 20
        IF (UPDATE(UPPTR) - MASTER(OMPTR,OMCOL)) 20, 30, 40
C------------- ВЗЯТЬ ЭЛЕМЕНТ ИЗ UPDATE; ПРОВЕРИТЬ НА ПЕРЕПОЛНЕНИЕ
   20 IF (NMPTR .GT. 1000) GO TO 99
        MASTER(NMPTR,NMCOL) = UPDATE(UPPTR)
        NMPTR = NMPTR+1
        UPPTR = UPPTR+1
      GO TO 10
C------------- ПРОПУСТИТЬ ЭЛЕМЕНТ В UPDATE
   30 UPPTR = UPPTR+1
      GO TO 10
C------------- ВЗЯТЬ ЭЛЕМЕНТ ИЗ СТАРОГО MASTER, ПРОВЕРИВ НА ПЕРЕПОЛНЕНИЕ
   40 IF (OMPTR .GT. OMSIZ) GO TO 50
        IF (NMPTR .GT. 1000) GO TO 99
        MASTER(NMPTR,NMCOL) = MASTER(OMPTR,OMCOL)
        NMPTR = NMPTR+1
        OMPTR = OMPTR+1
      GO TO 10
C------------- СЛИЯНИЕ ОКОНЧЕНО: ПРИСВ. ЗНАЧЕНИЯ OMCOL, UPSIZ И OMSIZ
   50 UPSIZ = NMPTR-OMSIZ-1
        OMSIZ = NMPTR-1
        OMCOL = NMCOL
      RETURN
C------------- ПРОИЗОШЛО ПЕРЕПОЛНЕНИЕ: ПРЕКРАТИТЬ РАБОТУ
   99 WRITE (PRT,100)
  100 FORMAT (' *** MASTER FILE SIZE EXCEEDED: HALTING IN MERGER ***')
      STOP
      END
```

Продолжение Рис. 0.11.

мой INPUT для чтения входных файлов. Рис. 0.10 иллюстрирует логику работы главной программы.

Теперь мы можем закончить проектирование двух подпрограмм и полностью запрограммировать решение поставленной задачи. Список параметров подпрограммы ввода INPUT должен содержать целый вектор для представления файла добавлений UPDATE, который нужно читать и передавать обратно в главную программу. Главная программа должна «знать» число шифров клиентов, которые были считаны подпрограммой INPUT; это число передается в главную программу как значение переменной UPSIZ. Наконец, если необходимо вместе со счетом печатать номер введенного списка шифров, то подпрограмма INPUT должна передать в главную программу и эту величину; мы ее назовем UPNUM. Таким образом, в главной программе вызов подпрограммы INPUT будет иметь вид, представленный на рис. 0.11.

Далее анализ подпрограммы MERGER показывает, что подпрограмма должна «знать» размеры основного файла и списка добавлений; назовем их соответственно OMSIZ и UPSIZ. Подпрограмма слияния должна «знать» также, какой столбец MASTER содержит текущий основной список; он будет назван OMCOL. Мы видим, что после слияния главная программа должна «знать», сколько добавилось новых клиентов, а также размер нового основного файла. Вместо того чтобы увеличить число параметров в списке возвращаемых величин, будем присваивать эти значения соответственно переменным UPSIZ и OMSIZ, поскольку предыдущие значения этих переменных больше не нужны.

На рис. 0.11 показано, как главная программа взаимодействует с подпрограммами.

Упражнения

0.01. Заданы два целых числа. Определить наибольшее целое число, на которое делятся оба числа без остатка (эта величина называется наибольшим общим делителем).

0.02. *Совершенное* число — целое, равное сумме его делителей. Например, 6 совершенное число, поскольку сумма его делителей 1, 2 и 3 равна 6. Напишите программу, которая читает произвольное количество целых чисел и выводит на печать те из них, которые являются совершенными.

0.03. Напишите программу учета товаров в магазине. Каждый учтенный предмет имеет 9-разрядный идентификационный код, розничную и закупочную стоимость товара. Программа должна следить за распродажей имеющихся товаров и закупкой новых следующим образом: положительные величины во вводном файле обозначают поступающие товары, и магазин должен их оплачивать; отрицательные величины обозначают проданные товары, оплаченные покупателями. Когда вводный файл исчерпан, делается полный учет имеющихся в наличии товаров и полученной прибыли.

0.04. Напишите программу, распечатывающую ежемесячный банковский отчет. В нем должны быть указаны все новые вклады и даты их поступления, а также ежедневный текущий баланс.

0.05. Вернитесь к подпрограмме MAX из разд. 0.7.5. Напишите аналогичную подпрограмму для работы с векторами, имя подпрограммы MAXV, ее параметры (X, N, Y, M, AMAX). Эта подпрограмма присваивает AMAX максимальное значение среди значений X и Y в интервалах от X (1) до X (N) и Y (1) до Y (M), где N, M$\leqslant$50.

0.06. Задан массив из N элементов, содержащий суммы вкладов. Напишите фрагмент программы, который печатает N+1 строк следующего вида:

```
                ТЕКУЩИЕ СУММЫ ВКЛАДОВ
ВКЛАД-1  СУММА  153.20 Р
ВКЛАД-2  СУММА  210.13 Р
и т. д.    и т. д.    и т. д.
```

0.07. Напишите подпрограмму FLIP, которая печатает элементы массива в обратном порядке. Например, применительно к приведенному ниже вектору X подпрограмма FLIP возвращает следующий результат:

```
Начальные значения X: 3, 4, 12, 9, 8
Результат работы FLIP (X): 8, 9, 12, 4, 3
```

Выполните эту работу только с одним вектором.

0.08. Допустим, что вам нужно определить лучшего автогонщика по результатам многих соревнований. Для каждого гонщика и каждого заезда вами подготовлена одна перфокарта, на которой записан номер гонщика, средняя скорость, показанная им в заезде, и место, которое он занял. В конце года вы сортируете перфокарты по номерам водителей. Напишите программу для определения первых пяти гонщиков в соответствии со средней скоростью по всем гонкам, в которых они финишировали. Водители, которые не выиграли ни одной гонки или принимали участие менее чем в пяти гонках, не должны рассматриваться программой.

0.09. Фирма SLOP (от SLOw Programming: программируйте медленно, но верно) заключила контракт на выполнение задания для фирмы ARRAYS INC. Фирма ARRAYS INC. производит продукцию самых различных типов. Задача программистов фирмы SLOP — написать программу, которая следит за продукцией фирмы ARRAYS INC. Каждый раз, когда произведена партия (или единица) продукции, оператор набивает на перфокарте значения переменных TYPE, QUANT, HOURS и COST, представляющих тип продукции, ее количество, время, затраченное на ее изготовление, и стоимость продукции.

Для каждого типа продукции может быть заведено несколько перфокарт, поскольку для ее производства может потребоваться работа людей на различных участках или в разное время дня. В конце каждой недели все перфокарты собираются. На число исходных перфокарт никаких ограничений нет, и они никак не отсортированы.

Нарисуйте блок-схему и напишите программу, выполняющую следующее задание. Для каждого типа продукции, изготовленной за день, вычислите общее затраченное время, стоимость продукции и общее количество единиц произведенной продукции. Хотя в течение дня может быть произведено не более 100 различных типов продукции, фирма ARRAYS INC. может производить тысячи различных деталей и каждая из них идентифицируется 9-разрядным числом. На печать следует выводить только информацию, относящуюся к данному дню. Еженедельная сводка должна состоять из пяти частей, по одной на каждый рабочий день недели. Перед набором рабочих перфокарт, полученных за день, установите перфокарту с датой по Юлианскому календарю (например, 77032 эквивалентно 2/1/77); для отличия сделайте эту величину отрицательной. На входе, таким образом, получится следующий поток данных: первая перфокарта с датой, затем колода рабочих перфокарт первого дня работы, вторая перфокарта с датой, за которой располагается колода рабочих перфокарт второго рабочего дня, и т. д. по пятый рабочий день включительно.

Пример заглавия и содержимого сводки:

```
* ARRAYS INC.   СВОДКА   ДАТА            82023
* ДЕТАЛЬ        ШТУК     ВРЕМЯ (ЧАС)          ОБЩ. СТОИМОСТЬ
  328           150      2.3                    800.58
  И Т. Д.       И Т. Д.  И Т. Д.                И Т. Д.
* ARRAYS INC.   СВОДКА   ДАТА            82024
                 И Т. Д.
```

0.10. Напишите программу игры в крестики-нолики на поле 3×3. Судья должен по очереди предоставлять ход двум игрокам. Ход заключается в том, чтобы поставить X или 0 в свободный квадрат. Судья объявляет победителем того, кто первым записал три крестика или нуля в ряд. Проверьте работу программы-судьи. При этом вы можете сами играть роль участников, указывая ваши ходы на терминале (в интерактивном режиме), или составить подпрограммы X и 0, которые будут делать ходы, т. е. решать, где расположить следующий маркер (представьте X и 0 целыми числами 1 и 0 соответственно).

0.11. Симметрическая матрица — это двумерный массив, в котором A (I, J)= =A (J, I) для всех значений I и J. Например, матрица

```
1 4 5
4 2 6
5 6 3
```

симметрическая и единственным образом представима с помощью треугольной матрицы:

```
1
4 2
5 6 3
```

Напишите подпрограмму SYM для проверки матриц произвольного размера менее 25. Подпрограмма должна выводить на печать нижнюю треугольную матрицу, если матрица оказалась симметричной, и возвращать логическое значение .TRUE. ИЛИ .FALSE. в зависимости от того, симметрична или несимметрична очередная матрица.

Глава 1

СТИЛЬ ПРОГРАММИРОВАНИЯ

Программирование сродни писательской деятельности. Любая программа, время жизни которой выходит за пределы одноразового выполнения, должна быть тщательно спланирована и построена, подобно тому как это делается при составлении пленарных докладов, статей, рекламных объявлений и т. д., т. е. всех средств письменного общения. Эту главу мы посвящаем стилю программирования.

Здесь и в последующих главах мы будем придавать большое значение рациональным методам, которыми пользуются квалифицированные программисты. Конечно, вместо того, чтобы просто запомнить все изложенные методы и пытаться применять их механически, вам следует подробно ознакомиться с этими методами, постараться понять, почему и как они применяются, и выработать ваш собственный стиль программирования, удовлетворяющий рассмотренным в книге критериям. Используемые нами методы не представляют собой каких-то жестких правил.

1.1. Пример

Мы начинаем данную главу с рассмотрения примера, в котором продемонстрировано много плохих приемов программирования. Мы придумали этот пример не только потому, что здесь мало места для всех реальных программ, в которых мы видели точно такие же дефекты. Одни программы были написаны студентами, другие представляют собой рабочие, часто выполняемые на ЭВМ программы или фрагменты больших систем.

На рис. 1.1 показана тривиальная программа инвентаризации в книжном магазине. Она работает с инвентарным файлом, в котором представлены номера книг и их количество, и обрабатывает запросы на закупку книг в соответствии с данным инвентарным файлом. Программа действительно работает (поразительно!). Теперь постарайтесь проанализировать, что делает программа и как.

В оставшейся части главы мы опишем некоторые ошибки, допущенные в программе. Вы также познакомитесь со структурой программы и ее отладкой. После этого вам, возможно, захочется

```
C
C     ПРОГРАММА BINGO - ВЕРСИЯ III, УРОВЕНЬ 8 ИЮЛЬ 17
C
      INTEGER A(100,2),BQ,CCL,AA(100,2)
      DATA A/200*0/,IPA/0/
C     СНАЧАЛА ВВЕСТИ ЗНАЧЕНИЯ NL И ND
      KE=0
         READ(5,107) NL,ND
C     НАЧАЛО ГЛАВНОГО ЦИКЛА DO
      DO 8 K=1,2
      IF (K.EQ.1) GO TO 84
107   FORMAT(I9,I9)
              IF(K.NE.1) GO TO 11
84       MN=NL
         GO TO 18
11       MN=ND
C     НАЧАЛО ВНУТРЕННЕГО ЦИКЛА DO
18    DO 3 IX=1, MN
      READ (5,204)  LN,  CCL
       IF (K-1) 16,43,70
 204  FORMAT(I8,I9)
C     СЛУЧАЙ А: НОВАЯ ПЕРФОКАРТА
43    A(IPA+1,1)=LN
      A(IPA+1,2) = CCL
      IPA=IPA+1
      GO TO 67
C     ПРОИЗОШЛА ОШИБКА
 16   GO TO 701
C     ОБРАБОТКА СТАРЫХ КАРТ
  70  DO 505 BQ=1,NL
      BBQ = 100-NL+BQ
      IF (A(BBQ,1).EQ.LN) GO TO 644
      GO TO 505
644      A(BBQ,K)=-CCL+A(BBQ,K)
505      CONTINUE
   67   GO TO  3
3        KK=K
      JJ=   100
      IF (KK.NE.2) GO TO 19
      WRITE (6,1919)A
1919   FORMAT(2X,7I10)
      STOP


C     ЭТО ДОЛЖЕН БЫТЬ ПЕРВЫЙ РАЗ; НЕ ОСТАНАВЛИВАЕМСЯ
19    DO 6 I=1,100
      JJJ=I+1
      IF((JJJ-1)/100.NE.KK) GO TO 153
      GO TO 6
174   JJ=JJ-1
      GO TO 702
153      DO 999 J=JJ J,JJ
      IF (.NOT.A(J,1).LT.A(I,1)) GO TO 302
      AA(I,2)=A(J,2)
      A(J,2)=A(I,2)
      A(I,2)=AA(I,2)
C     ПРОВЕРИМ: МЕНЬШЕ ЛИ A(I,1), ЧЕМ A(J,1)
 302  IF (A(I,1).LT.A(J,1)) GO TO 17
      AA(J,1)=A(I,1)
      A(I,1)=A(J,1)
      A(J,1)=AA(J,1)
17    IF (I.LT. J) GO TO 999
C     ПРОИЗОШЛА ОШИБКА
      GO TO 703
999      CONTINUE
    6 C O N T I N U E
      GO TO 8
703   KE=KE+1
702   KE=KE+1
701      KE=KE+1
      WRITE(6,704) KE
704   FORMAT(' ERROR; KE=',I5)
8     CONTINUE
      STOP
      END
```

Рис. 1.1. Пример — программа инвентаризации книг. (Текст в инстр. 704: «Ошибка».)

переделать программу. Однако мы не будем исправлять приведенную программу, чтобы сделать ее хорошей. Мы надеемся, что она просто послужит вам серьезным уроком. Иногда лучше выбросить плохую программу, чем пытаться ее исправить.

1.2. Документация

Когда вы встречаетесь с термином «документация», вы, вероятно, полагаете, что речь идет о комментариях внутри программы. Это правильный, но только неполный ответ. Существует внутреннее и внешнее описание программы. «Внешняя документация» необходима для распространения программ между пользователями, которые реальную исходную программу могут никогда не увидеть. Внешняя документация удобна также для описания программы в тех случаях, когда размеры описания слишком велики для того, чтобы уместить его в комментариях. Комментарии (часть «внутренней документации») дают всевозможные пояснения к программе, помогают пользователю (да и самому разработчику) анализировать исходный текст программы.

1.2.1. Внешняя документация

Внешняя документация предназначена для двух различных групп людей: для тех, кто пользуется программами, и тех, кто занимается их эксплуатацией. Внешняя документация для пользователей программ содержит информацию, позволяющую работать с данной программой. Такая документация называется *руководством для пользователей* и содержит следующие сведения:

1. Имя программы.
2. Краткое описание того, что делает программа.
3. Форма и смысловое значение данных, вводимых в программу.
4. Вывод из программы и способы управления объемом вывода или типом выводимых данных.
5. Необычные ситуации в программе, ограничения и недостатки применяемого метода.
6. Пример результатов работы программы, показывающий в полном объеме входные данные и соответствующий им выход.
7. Сведения об авторах программы или специалистах по ее эксплуатации для контакта с ними в случае ошибочных результатов.
8. Благодарности и ссылки на литературу.

В одних программах какие-то из перечисленных сведений необходимо изложить более подробно, в других нужно уделить

больше внимания чему-то еще. С приобретением опыта вы сможете сами решать, как лучше документировать программы.

Теперь давайте сравним то, что предлагается, с тем, что имеется в приведенном нами плохом примере. Данному примеру предшествовал один абзац пояснений. Если бы было нужно воспользоваться этой программой, то нам было бы неизвестно ни как подготовить для программы входные данные, ни как интерпретировать вывод. Таким образом, вы уже можете вполне оценить значение руководства для пользователя.

Недостатки внешнего описания приведенной программы даже более существенны для того, кто ее эксплуатирует. Если вам понадобилось бы расширить эту программу или откорректировать какие-либо неверные результаты, это было бы сделать непросто. К сожалению, эта программа — типичный представитель «производственных» программ: она работает правильно при «стандартных» входных данных, т. е. на тестовых данных автора и первом наборе реальных данных, на которых программа выполнялась. Однако существует множество наборов данных, на которых программа не будет работать правильно. В конце концов одна из скрытых ошибок проявится. Если к вам обратятся с просьбой исправить данную программу, неизвестно даже, с чего начать поиск ошибки.

Внешняя документация для обслуживающего персонала часто называется *руководством по логике программы* (Program Logic Manual, сокращенно PLM), поскольку в нем объясняется логическая структура программы. Такое руководство обычно содержит следующие разделы:

1. Назначение программы.
2. Имена и назначение модулей программы.
3. Последовательность вызова модулей — список модулей, вызываемых каждым модулем.
4. Имена и смысл основных переменных.
5. Ход выполнения программы — последовательность действий во время выполнения программы.
6. Описание средств, способствующих отладке программы, и способы их использования.
7. Расширения, запланированные автором программы, и методика связи этих расширений с оставшейся частью программы.

И в данном случае перечисленная информация различна для различных программ. Вы должны представить себя в роли потенциального пользователя программы или обслуживающего ее программиста и попытаться предвосхитить вопросы или проблемы, которые возникнут перед теми, кто будет работать с вашей программой.

Мы говорили о руководстве для пользователей и о руководстве по логической структуре программы так, как если бы они были составлены после написания программы. Хотя конечная шлифовка документации делается *постфактум*, содержание этих руководств представляет собой важную и неотъемлемую часть построения программы. И наконец, если описания помогут кому-нибудь другому понять или использовать исходный текст программы, разве они не помогут вам сделать то же самое в процессе создания или отладки программы?

Возьмем, например, имена модулей, параметров и их назначение. Это, безусловно, важная часть проектирования программы. Обычно вначале мы упрощаем проблемы, связанные с деталями работы каждого отдельного модуля, и занимаемся лишь принципиальной схемой их взаимодействия. Мы предпочитаем в первую очередь сосредоточить внимание на основных моментах работы программы, а затем уже переходим к рассмотрению деталей. (Такой метод построения программы называется пошаговой разработкой. Мы воспользуемся позднее этим методом в разд. 1.9, чтобы заново переделать приведенную выше программу.)

1.2.2. Внутренняя документация

Автор приключенческого романа должен проделать две вещи. Прежде всего он должен описать время и место действия романа, обозначить интригу и представить своих героев. После того как читатель разберется в обстановке, можно вести изложение, следуя сюжетной линии. Аналогичная последовательность имеет место не только в художественном произведении. Эта книга, например, началась с вводной главы, чтобы все читатели имели определенную минимальную степень подготовки; после вводных замечаний был представлен новый материал.

При написании программ для ЭВМ необходимо следовать тем же правилам. Прежде всего вы должны представить вашу программу: что это за программа, как (обычно) она работает, какие имеет входы и выходы, какие ключевые переменные? После такого введения вы можете продолжить построение программы, зная, что любой читатель сможет проследить за тем, что вы намереваетесь сделать и приблизительно как вы это сделаете.

По этой причине мы разделим внутреннюю документацию на две части: «шапку» или «блок комментариев», располагаемый в верхней части программы, и «пошаговые комментарии» внутри программы.

Мы будем выносить обычно следующую информацию в блок комментариев:

1. Имя программы и ее более подробное название.
2. Имя программиста.

3. Дату (и номер версии, если программа существенно изменена).
4. Сжатое изложение того, что должно быть сделано, краткий обзор используемого алгоритма, план порядка действий.
5. Ожидаемые входные данные.
6. Имена и назначение основных переменных.
7. Вызываемые подпрограммы.
8. Используемые нестандартные или системно-зависимые средства (их следует по возможности избегать).
9. Необычные условия или аварийные ситуации, которые могут возникнуть при выполнении программы.

Сравните эти положения с «шапкой» программы на рис. 1.1. Покажем теперь, какой вид имело бы описание, если бы мы составили его в соответствии с вышеизложенными рекомендациями:

ПРОГРАММА BINGO — BOOK INVENTORY GENERATOR
ИНВЕНТАРИЗАЦИИ КНИГ

ПРОГРАММА СЛЕДИТ ЗА СПИСКОМ НОМЕРОВ КНИГ И ИХ КОЛИЧЕСТВОМ.

ПРОГРАММА ДЕЛИТСЯ НА ЧЕТЫРЕ ФАЗЫ:

1. ЧТЕНИЕ ПЕРФОКАРТ, СОДЕРЖАЩИХ НОМЕРА КНИГ, ИМЕЮЩИХСЯ В НАЛИЧИИ, И ИХ КОЛИЧЕСТВО;
2. СОРТИРОВКА ИНВЕНТАРНЫХ ДАННЫХ В ВОЗРАСТАЮЩЕМ ПОРЯДКЕ ПО НОМЕРАМ;
3. ЧТЕНИЕ ПЕРФОКАРТ, СОДЕРЖАЩИХ НОМЕРА И КОЛИЧЕСТВО ПРОДАННЫХ КНИГ;
4. ПЕЧАТЬ ОТЧЕТА ОБ ИМЕЮЩИХСЯ В НАЛИЧИИ КНИГАХ И ИХ КОЛИЧЕСТВЕ.

ВХОДНЫЕ ДАННЫЕ:

1. ПЕРФОКАРТА-ЗАГОЛОВОК, УКАЗЫВАЮЩАЯ, СКОЛЬКО ПЕРФОКАРТ БУДЕТ СЧИТАНО В ФАЗЕ 1 И ФАЗЕ 3 (ПО ОДНОМУ НОМЕРУ КНИГИ НА ПЕРФОКАРТЕ)
2. ПЕРФОКАРТЫ С НОМЕРАМИ ИМЕЮЩИХСЯ В НАЛИЧИИ КНИГ И ИХ КОЛИЧЕСТВОМ ДЛЯ ФАЗЫ 1
3. ПЕРФОКАРТЫ С НОМЕРАМИ ПРОДАННЫХ КНИГ И ИХ КОЛИЧЕСТВОМ ДЛЯ ФАЗЫ 3

ОСНОВНЫЕ ПЕРЕМЕННЫЕ:

NBOOKS — СКОЛЬКО ПЕРФОКАРТ «КНИГА — КОЛИЧЕСТВО» ДЛЯ ФАЗЫ 1
NSALES — СКОЛЬКО ПЕРФОКАРТ «КНИГА — ПРОДАННОЕ КОЛИЧЕСТВО» ДЛЯ ФАЗЫ 3
BOOKNO — МАССИВ НОМЕРОВ КНИГ
QTY — ИМЕЮЩЕЕСЯ В НАЛИЧИИ КОЛИЧЕСТВО — QTY(I) ЭТО КОЛИЧЕСТВО BOOKNO(I)
CBOOK — ТЕКУЩИЙ ОБРАБАТЫВАЕМЫЙ НОМЕР КНИГИ
NENTRY — ЧИСЛО ЭЛЕМЕНТОВ В МАССИВАХ BOOKNO И QTY

ПРОГРАММИСТ: ——— ДАТА:———

Очевидно, что такое описание сообщает гораздо больше информации, чем одиночный комментарий — заголовок в исходной программе. (Заметим также, что данное описание дает ключ к тем структурным изменениям, которые мы в дальнейшем произведем.)

Теперь проанализируем оставшиеся описания внутри нашей программы-примера. Два комментария информируют о циклах DO (которые вы, конечно, заметили бы и без комментария), два комментария предупреждают об ошибке, которая может произойти, не объясняя источника или типа ошибки. Другие комментарии говорят что-то о «новых картах», «старых картах» или «первом разе». Короче, комментарии внутри программы не дают читателю практически никакой информации.

По нашему мнению, следующие рекомендации способствуют эффективному документированию:

1. Используйте комментарии, которые описывают назначение сегмента программы, не повторяя информации, содержащейся в самой программе. Такой комментарий, как «ПРИБАВИТЬ 1 К СЧЕТЧИКУ», предшествующий COUNT=COUNT+1, намного менее полезен, чем «УВЕЛИЧИТЬ СЧЕТЧИК ПРОЧИТАННЫХ КАРТ С ДАННЫМИ».

2. Не комментируйте каждую инструкцию. Цель комментария — объяснить характер действий в программе. В каждой отдельной инструкции довольно редко проявляются какие-либо действия, и поэтому нет необходимости описывать каждую инструкцию отдельным комментарием. Комментарии должны помогать следить за выполнением программы.

3. Используйте комментарии для разделения программы на отдельные подразделы. Такие подразделы содержат обычно от 5 до 10 инструкций и имеют одноцелевое назначение.

4. Используйте больше комментариев при разделении программы на крупные фазы. Такая фаза представляет собой одноцелевой блок, как правило, длиной не более страницы.

5. При расположении комментария часто бывает полезно делать отступ от текста программы. Так, для инструкций, начинающихся в колонке 7, комментарии можно начинать примерно с колонки 15. Такая отбивка позволяет читать отдельно комментарии и текст программы.

Вы можете обратиться к программе, приведенной в конце гл. 0, чтобы сравнить документацию этой программы с изложенными здесь положениями. Сейчас мы не будем показывать комментарии, которые следовало бы использовать в примере про-

граммы, приведенном в настоящей главе. Однако, когда мы займемся изменением этой программы и в других аспектах, мы включим комментарии, придерживаясь изложенных здесь стандартов. Окончательно откорректированная программа приведена в разд. 1.9.

1.2.3. Дополнительные рекомендации

До сих пор мы описывали внутреннюю документацию исключительно в терминах комментариев. Но точное значение глагола «документировать» означает давать информацию. Поэтому, даже если внутри программы что-либо сделано в этом плане, то это является частью документации. Нижеследующие советы помогут сделать вашу программу более выразительной.

1. Выбирайте осмысленные имена. X=Y/I дает гораздо меньше информации, чем AVG=SUM/N.

2. Не забывайте правило I—N для обозначения целых переменных. Программисты давно уже привыкли к нему и автоматически считают, что переменные, начинающиеся с I, J, K, L, M, N,— это целые. Чтобы помочь читателю вашей программы, следуйте этому соглашению во всех возможных случаях.

а) Имена I, J, K и N должны всегда обозначать целые. (По соглашению I, J и K должны использоваться также только в качестве счетчиков индексов циклов или массивов индексов.)

б) Однако в некоторых случаях не следует ломать осмысленные имена, с тем чтобы они соответствовали стандартному соглашению. Имена вещественных переменных, такие, как MILES или MEDIAN, *не* следует заменять на XMILES или EMDIAN только для того, чтобы подогнать под это правило. Цель состоит в том, чтобы улучшить наглядность программы, но не делать ее неясной. Трудно улучшить такие имена, как MILES или MEDIAN.

3. Придерживайтесь очевидного порядка действий. Если надо вычислить

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

то

```
DSCRM = SQRT(B**2 - 4*A*C)
ROOT(1) = (-B+DSCRM) / (2*A)
ROOT(2) = (-B-DSCRM) / (2*A)
```

конечно, намного понятнее, чем

```
   J=1
   DSCRM=(B*B-4*A*C)**0.5
10 ROOT(2-(J+1)/2) = (-B + J*DSCRM) / (2*A)
   J=-J
   IF (J .LT. 0) GO TO 10
```

4. Прерывайте длинные инструкции на доступные для понимания фрагменты; избегайте, однако, излишне мелкого дробления. Во-первых, человеческий мозг может помнить активно в любой конкретный момент времени около пяти «единиц» информации. Поэтому за сложными выражениями, состоящими более чем из пяти нетривиальных частей, нелегко проследить. Во-вторых, чем длиннее выражение, тем больше потенциальных возможностей для появления лишних скобок, неправильных знаков операций и ошибочных переменных. Целесообразно поэтому разделять длинные инструкции на части, за которыми можно проще следить и которые легче отлаживать. Вышеприведенное выражение, вычисляющее дискриминант квадратного уравнения, является примером такого разделения длинного выражения.

5. Аналогично используйте дополнительные скобки и пробелы, чтобы яснее подчеркнуть ваши намерения.

Выражение вида

```
IF (A+(B*C) .GT. 8 .OR. (I .LT. 5 .AND. J .GT. 2)) GO TO 10
```

понять легче, чем

```
IF (A+B*C.GT.8.OR.I.LT.5.AND.J.GT.2) GO TO 10
```

6. Использование отступов в тексте программы является мощным средством, позволяющим сделать программу более выразительной. Квалифицированные программисты, работающие на языках ПЛ/I и Алгол, уже в течение ряда лет используют этот метод, но программирующие на Фортране только сравнительно недавно отошли от привычки начинать каждую инструкцию с седьмой колонки. Целесообразно писать с отступом DO-циклы и IF-группы. Рассмотрим следующий пример:

```
      DO 200 HOURNO = 1, 24
         WRITE(IOUT,100) HOURNO
100      FORMAT(' PROGRESS REPORT FOR HOUR',I4)
         DO 150 I = 1, N
            SUM=SUM+X(I)
            IF (X(I).LE.10) GO TO 120
               X(I)=X(I)-10.
               GO TO 150
120         WRITE(IOUT,125) I, X(I)
125         FORMAT(' SAMPLE ', I2,' HAS VALUE ', F10.4)
            X(I)=0.
150      CONTINUE
         WRITE(IOUT,175) SUM
 75      FORMAT(' TOTAL SUM FOR THIS HOUR IS', F12.4)
         SUM=0.
200   CONTINUE
      STOP
      END
```

Даже не понимая сущности задачи, легко видеть структуру этой программы. Следующие рекомендации, по нашему мнению, полезно соблюдать при оформлении текста программы:

(а) Тело цикла следует начинать на три пробела правее его первой и последней инструкций. Это относится как к обычным, так и к «самодельным» циклам DO (т. е. циклам, в которых по ряду причин необходимо использовать инструкции IF и GO TO).

(б) Следующие за IF инструкции, выполняемые при условии «ложно», можно также располагать на три пробела правее начала инструкции IF, если этих инструкций больше чем одна. Например,

```
      IF (X(I) .LE. X(I+1)) GO TO 40
         TEMP = X(I)
         X(I) = X(I+1)
         X(I+1) = TEMP
   40 CONTINUE
```

(в) Можно расположить правее седьмой колонки одну или обе группы инструкций в конструкции IF-THEN-ELSE. (Конструкция IF-THEN-ELSE описана в разд. 1.3.2. Это просто форма инструкции IF, в которой выполняются два различных действия в зависимости от проверяемого условия «истинно» или «ложно»). Ниже приведен пример расположения текста в конструкции IF-THEN-ELSE:

```
      IF (HOURS .GT. 40.) GO TO 450
         PAY = RATE*HOURS
         TAX = 0.23*PAY
      GO TO 500
  450    PAY = RATE*HOURS + 0.5*RATE*(HOURS-40.)
         TAX = .23*PAY + .08*RATE*(HOURS-40.)
         NOVRT = NOVRT+1
  500 NEMP = NEMP+1
```

(г) Удобнее читать DO — циклы и большинство самодельных циклов, если цикл оканчивается инструкцией CONTINUE.

Мы советуем использовать метод отступов, чтобы сделать программу нагляднее. Странно, но факт — программы, для которых вы нашли время тщательно расположить текст, легче поддаются отладке.

7. Используйте последовательные, логически связанные номера инструкций. Если вы встречаете инструкцию GO TO 675 в середине программы, а последний номер инструкции, который вы видели, был 403, то вы сочтете, что номер 675 расположен ниже в программе. Квалифицированные программисты используют номера инструкций (наряду с комментариями, конечно) для информирования о начале новой секции программы. Каждая сек-

ция имеет свой собственный диапазон номеров инструкций (например, от 1000 по 1999, от 2000 по 2999 и т. д.), и все номера инструкций внутри такой секции принадлежат этому диапазону. Тогда, если секции делятся на подсекции, то каждая подсекция составляет часть диапазона секции (с номерами инструкций от 1000 по 1199, от 1200 по 1399 и т. д.). При таком подходе легко находить номера инструкций и следить за связанными блоками программы.

8. Избегайте имен переменных и номеров инструкций, которые легко спутать, например такие имена, как MEAN и MEEN или II и I1, номера инструкций 1883 и 1383 или 9099 и 9909. Они могут вызвать трудности при написании программы; катастрофические результаты из-за невинных ошибок в набивке; могут сделать программу трудной для чтения.

В итоге всегда следует иметь в виду, что кто-то, может быть, даже и вы, будет читать вашу программу. Эксплуатация программ — задача не тривиальная. Если вам понадобится вернуться вновь к вашей программе спустя год после ее написания, или кто-нибудь другой попытается понять логику работы вашей программы, или вам будет необходимо понять логику работы чьей-либо еще программы, вы скажете спасибо, если программа легко читается.

1.3. Структура программы

Вероятно, вы уже в состоянии написать некоторые довольно сложные программы (длиннее 30 инструкций). В коротких программах различие в структурах обычно оценить трудно из-за недостаточной сложности реализуемого алгоритма. Структуры программ, как и документация, отличаются у различных программистов и в различных проектах. Мы хотим, чтобы вы прочитали этот раздел и поняли, почему мы рекомендуем описанные здесь методы.

1.3.1. Дискуссия об инструкции GO TO

В 1968 г. известный ученый Едсгер Дейкстра написал статью, в которой, в частности, отметил, что использование инструкции GO TO затрудняет чтение программ. Справедливость этого заявления бесспорна для всех, кто имел дело с большим количеством фортранных программ. Суть его предложения состояла в том, чтобы подтолкнуть разработчиков будущих языков программирования к созданию достаточно богатого словаря структур управления, с тем чтобы отпала необходимость в инструкции GO TO. Это заявление Дейкстры привлекло достаточно много внимания, чтобы рассмотреть подробнее его мотивировку.

Дейкстра исходил из того, что язык должен содержать средства для управления циклами, для выполнения альтернативных действий, зависящих от одного или нескольких условий, для выбора одной из групп инструкций, которые должны выполняться в том или ином «подслучае».

Проблема состоит в том, что язык Фортран применяется уже на протяжении более 20 лет. Он обновлялся несколько раз, но реализация предложений, подобных высказанным Дейкстрой, привела бы к существенному изменению языка. При каждом изменении Фортрана сохранялась преемственность — максимально возможная совместимость «вверх»: программа, которая компилировалась в старом стандарте языка, должна была компилироваться и в новом стандарте. (Конечно, расширения и новые средства языка означают, что не все программы, компилирующиеся в новом стандарте, могут компилироваться в старом.) По существу, мы не можем ожидать каких-либо принципиальных изменений в Фортране; необходимо научиться разумно пользоваться тем, что уже существует.

Люди часто ищут легких, автоматических путей решения проблем. Проще всего сделать такое следствие из заявления Дейкстры: если программа, содержащая инструкции GO TO, плоха, то программа без инструкций GO TO должна быть хорошей. Это неверно! Используя сложные вложенные DO-циклы и вызовы подпрограмм, скудные пояснения, нелогично упорядоченные инструкции и другие подобные хитрости, можно написать совершенно непонятную программу, но без инструкций GO TO. Таким образом, программа, в которой нет GO TO, не будет непременно хорошей программой.

На Фортране можно написать хорошую программу, несмотря на то что Фортран имеет ограниченный набор структур управления. Даже не обладая изощренными инструментами, мастер может создать прекрасную вещь.

Одним из достоинств Фортрана является его стандартизованность; это означает, что программы, написанные на стандартном Фортране, можно выполнять на самых различных машинах. Более того, Фортран — сравнительно простой язык; поэтому компиляторы Фортрана относительно небольшие и в связи с этим доступны для большого числа мини-ЭВМ. Фортран — широко распространенный язык; может оказаться, что вы просто найдете необходимую вам программу, и ее не нужно будет писать самому. Существует большое число хорошо проверенных компиляторов; некоторые из них имеют специальные средства, такие, как вспомогательные средства отладки или оптимизация объектного кода. Наконец, что самое главное в этой дискуссии, желаемые средства управления можно смоделировать в стандартном Фортране зачастую без существенной потери эффективности и

удобочитаемости. Таким образом, язык Фортран полностью удовлетворяет требованиям, которые предъявляются к языку высокого уровня при решении по-настоящему сложных задач.

Специалисты разработали расширенные или «структурные» процессоры Фортрана. Обычно такие процессоры имеют форму предпроцессора, который воспринимает на входе программы расширенной версии Фортрана и создает в качестве выхода эквивалентную стандартную программу на Фортране; эта программа вводится в компилятор Фортрана, который генерирует объектный код.

Указанные процессоры не стандартны. Более того, существует много процессоров, но нет соглашения, какие расширения обязательны. Создано много различных диалектов «структурного Фортрана». Если вы будете пользоваться таким процессором, то сузится круг машин, на которых можно выполнять вашу программу. Программу можно будет выполнять лишь на той машине, на которой вы работаете, или на той, которая имеет совместимый процессор. При этом полностью теряется одно из главных преимуществ Фортрана — мобильность (портативность) написанных на нем программ.

С другой стороны, упомянутые выше процессоры могут позволить программировать алгоритмы в более «естественном» синтаксисе. Однако само по себе это еще не улучшает программ точно так же, как не улучшает их автоматическое исключение инструкций GO TO. Если вы располагаете предпроцессором и вас устраивают расширения, которые он обеспечивает, возможно, вы захотите воспользоваться таким процессором. Его применение, правда, несколько увеличит время компилирования программы. Мы придерживаемся, однако, такого мнения, что немного машинного времени гораздо менее ценно, чем длительное время работы программиста. Если такие средства могут помочь вам программировать быстрее или с меньшим числом ошибок, дополнительные затраты на машинное время будут оправданы.

1.3.2. Структуры управления

В этом разделе мы покажем несколько структур, используемых в хороших алгоритмах. Грамотные программисты применяют такие конструкции уже в течение длительного времени. Проанализируйте их и запомните, они пригодятся вам при решении ваших задач. Вы найдете, мы в этом уверены, что алгоритмы решения многих задач могут быть естественно выражены с использованием изложенных ниже конструкций.

1. IF-THEN. Если условие удовлетворяется, то выполнить определенную группу инструкций. Эту форму можно применять в таких случаях, как сортировка (если два элемента расположе-

ны не по порядку, то поменять их местами); системах инвентаризации (если существует достаточное количество товара x, то обработать запрос на продажу количества y товара x); при моделировании (если надо реализовать еще какие-то события, то взять первое событие и реализовать его). Эффект достигается логической инструкцией IF, как показано ниже:

```
      IF (NAME(I).LE.NAME(I+1)) GO TO 50
         TNAME = NAME(I)
         NAME(I) = NAME(I+1)
         NAME(I+1) = TNAME
50 I = I+1
```

2. IF-THEN-ELSE. Если условие удовлетворяется, то будет выполнена одна группа инструкций программы; в противном случае выполняется другая группа инструкций. Обе альтернативные группы часто вновь сходятся на общей для обеих групп инструкции. Эта конструкция представлена в Фортране или двумя инструкциями IF, или одиночной инструкцией IF и парой инструкций GO TO. В структуре с двумя инструкциями IF одна инструкция проверяет требуемое условие, вторая — обратное условие (логическое .NOT.). При использовании одной IF и двух инструкций GO TO программа имеет вид

```
IF    (УСЛОВИЕ) GO TO SN1
      ИНСТРУКЦИИ, ВЫПОЛНЯЕМЫЕ, ЕСЛИ УСЛОВИЕ ЛОЖНО
GO    TO SN2
SN1   ИНСТРУКЦИИ, ВЫПОЛНЯЕМЫЕ, ЕСЛИ УСЛОВИЕ ИСТИННО
SN2   СЛЕДУЮЩАЯ ИНСТРУКЦИЯ
```

3. Циклы. Применяются, когда требуется повторное выполнение ряда действий в зависимости от условия. Может быть задано условие на определенное число повторений, или пока справедливо определенное утверждение, или пока не произойдет определенное событие, условие может быть некоторой комбинацией других условий. С этой формой мы встречаемся, например, при приближенном вычислении квадратного корня, если указана требуемая точность или требуемое число итераций; при поиске в таблице некоторого конкретного значения, который продолжается до тех пор, пока это значение не будет найдено или не будет просмотрена вся таблица. Обычно такую конструкцию можно реализовать инструкцией DO. В языках ПЛ/I, Алгол, Кобол, Паскаль она действительно реализуется единственной инструкцией цикла. Но даже в Фортране 77 инструкция DO так ограничена, что сложные циклы часто реализуются с использованием инструкций IF и GO TO. Циклы IF/GO TO применяются при наличии двух необходимых независимых условий останова, при постоянном шаге приращения, когда нужно проверить условие завершения цикла *перед* началом его выполнения, а не после. Существует два классических типа циклических

инструкций: с проверкой окончания цикла *до* и *после* выполнения тела цикла. Инструкции цикла имеют также названия DO-WHILE и REPEAT-UNTIL.

4. Выбор варианта. Применяется, когда нужно перейти на ту или иную группу исполняемых инструкций в зависимости от значения переменной. Все группы инструкций передают управление в общую точку, расположенную после последней группы. В качестве примера применения такой конструкции может служить программа, следящая за книгами в библиотеке; такой программе требуется выбор одного из нескольких действий, а именно: приобретение новой книги, выдача книги, возврат книги, изъятие книги из обращения, «координаты» нужной книги, имя читателя книги и т. п. В некоторых языках эта конструкция реализована непосредственно, но в большинстве языков программист должен ее смоделировать, используя вычисляемый GO TO. При небольшом числе возможных вариантов конструкции IF-THEN-ELSE, по-видимому, предпочтительнее, чем вычисляемый GO TO. Важно иметь хорошую документацию при работе с этой формой. В программе, использующей вычисляемый GO TO, инструкция выбора варианта имеет следующий вид:

```
C GO TO (SN1, SN2, . . ., SNM), ДЕЙСТВИЕ
C                    ПЕРВЫЙ СЛУЧАЙ
  SN1 ФРАГМЕНТ, ВЫПОЛНЯЮЩИЙСЯ В СЛУЧАЕ 1
      . . .
      GO TO SNX
C                    ВТОРОЙ СЛУЧАЙ
  SN2 ФРАГМЕНТ, ВЫПОЛНЯЮЩИЙСЯ В СЛУЧАЕ 2
      . . .
      GO TO SNX
        . . .
C                    ПОСЛЕДНИЙ СЛУЧАЙ
  SNM ФРАГМЕНТ ДЛЯ ПОСЛЕДНЕГО СЛУЧАЯ
      . . .
      GO TO SNX
C
C                ОБЩАЯ ТОЧКА ВСЕХ ГРУПП
  SNX ОСТАЛЬНАЯ ПРОГРАММА
```

5. ПОДПРОЦЕДУРЫ. Отдельный блок программы для выполнения специфических действий. Такая конструкция полезна, (а) когда должны быть выполнены примерно одни и те же действия в нескольких местах программы, возможно, с разными наборами данных; (б) когда относительно независимый набор операций может быть выделен из оставшейся части программы, чтобы ее было удобнее читать; (в) когда сегмент программы может

быть отдельно написан и отлажен. Обычно это реализуется в Фортране с помощью подпрограммы или функции.

Приведенные пять структур управления — это основные структуры, которые применяют квалифицированные программисты. Мы убедительно просим вас изучить эти структуры и использовать их как можно чаще при решении ваших задач. Может показаться слишком сильным утверждение, что все задачи поддаются решению при использовании указанных конструкций, однако, по-видимому, естественное решение большинства задач укладывается в эти рамки.

В разд. 1.2 вы убедились, какое важное значение имеют комментарии при подготовке программы для тех, кто с ней работает. Аналогично комментариям приведенные структуры управления помогают читателю понять вашу программу. Прежде чем читатель начнет просматривать некий фрагмент программы, ему полезно узнать, что этот фрагмент является частью структуры цикла. Заметим также, что инструкции в рассмотренных структурах выполняются по принципу «сверху вниз». Все переходы (GO TO) передают управление к точкам, расположенным ниже в теле программы, за исключением структур цикла. В структурированных программах управление организовано именно таким образом.

1.4. Защита от ошибок при программировании

Во всех без исключения достаточно сложных программах есть ошибки. Ошибки могут возникнуть от разных источников, например от непонимания задачи, плохой программы, набивки инструкций с ошибками. Обычно над такими источниками ошибок имеет контроль сам программист.

Другие потенциально возможные источники ошибок находятся вне сферы действий программиста, например ошибки во входных данных. В коммерческих системах, когда первоначальная отладка программы завершена, программист передает программу потребителям и не может предотвратить такие ошибки, которые возникают при вводе неправильно набитых данных, при неверном расположении или пропуске данных, при отсутствии карт с признаком конца данных.

В программе следует предусмотреть защиту от ошибок во входных данных. Программу можно защитить от ошибок, руководствуясь следующими соображениями:

1. Минимизируйте ограничения на ввод. Пользуйтесь специальной перфокартой для сигнализации о конце входной колоды перфокарт вместо заглавной перфокарты, содержащей количество вводимых перфокарт. В подсчете перфокарт легко оши-

биться. Если ваш компилятор позволяет, то лучше применять операнд END= в инструкции READ, чем пользоваться перфокартой с признаком конца колоды.

2. Следите за допустимыми значениями входных данных. Во многих случаях существует четкий диапазон входных значений. Например, число рабочих часов служащего в неделю должно находиться в диапазоне между 0 и 7*24 часа, а вес человека — между 1 и 200 кг. В этом смысле программа должна проверять каждое входное значение.

3. Для контроля выводите входные данные на печать. Следует печатать все считываемые данные непосредственно после ввода; такой вывод должен быть четко оформлен. Указанный метод поможет вам или пользователю точно проследить по листингу, какие значения введены в программу (что особенно полезно, если какие-то данные были набиты на перфокартах ошибочно). При отладке программы в этом случае вы будете располагать значениями всех данных, которые были введены в программу, и сможете проконтролировать работу программы вручную.

4. Во-время останавливайте выполнение программы. Если программа ожидала, но не получила входные данные, следует напечатать сообщение об ошибке и прекратить выполнение программы. Точно так же, если вы ожидали получить ненулевое значение, а получили в результате нуль (возможно, из-за того, что в соответствующей колонке перфокарты ничего не было пробито), не следует продолжать выполнение программы и порождать ошибочные результаты. В общем всякий раз, когда программа сталкивается с неприемлемой ситуацией любого вида, следует печатать сообщение об ошибке и прекращать выполнение.

1.5. Методы отладки

Программисты кажутся невероятными оптимистами. Начинающие программисты считают, что если в программе нет синтаксических ошибок, то программа правильна. Опытные программисты допускают наличие семантических ошибок. Однако, несмотря на совершенную очевидность противного, они полагают, что каждая программа будет правильно выполняться с первого раза. Начинающие программисты, даже допуская, что вначале их программы не будут свободны от ошибок, все же думают, что, когда они наберутся опыта, эта ситуация изменится. Но это ошибочно. Требуется очень большой опыт для понимания того, что ошибки программирования всегда будут иметь место и что нужно заранее принять меры, облегчающие, насколько это возможно, их обнаружение.

1.5.1. Средства отладки

Отладка — это исследовательская, экспериментальная работа. Важно знать методы отладки и научиться успешно применять их. Ниже описаны некоторые средства отладки, которые мы находим полезными.

1. Изучите доступные вам средства отладки.

Многие компиляторы допускают расширения Фортрана, которые в случае ошибок в программе дают дополнительную информацию пользователю. Компилятор WATFIV, например, обрабатывает инструкцию DUMPLIST, а для Фортрана-10 фирмы DEC имеется пакет FOR-DDT, и даже «промышленный» компилятор с Фортрана G фирмы IBM позволяет использовать средства отладки (DEBUG). Мы не будем вдаваться в подробности каждого из указанных средств отладки, поскольку они содержат массу всевозможных особенностей. Часто обеспечиваемые ими услуги включают:

(а) Завершающий дамп. В случае ненормального окончания задания (а иногда и при нормальном окончании) на печать выводятся текущие значения всех или выбранных групп переменных.

(б) Трассировку выполнения инструкций программы. Программист указывает границы или ключевые точки. Каждый раз, когда управление попадает в пределы установленных границ, создается «карта» прохождения, которая идентифицирует все инструкции, выполняющиеся внутри данной области. Часто ход выполнения программы представлен в виде номеров исполняемых инструкций или последовательности передач управления: «Выполнены — инструкции с номерами 1, 2, 3, 4, 5, 6, 4, 5, 6, 4, 5, 6, 7, 8» или «Выполнение началось с номера 1. Переход с 6 на 4. Переход с 6 на 4. Окончание на 8». Последняя форма предполагает последовательное выполнение инструкций с номерами 1, 2, 3,...; регистрируются только переходы (с 6 на 4).

(в) Контроль за определением переменных. Переменные, которым еще не присвоены значения (инструкцией инициализации данных DATA, входными данными, инструкцией присваивания или каким-либо другим способом), называются «неопределенными». Каждая переменная связана со словом в памяти, а каждое слово содержит некоторое значение. Если программа не поместила в слово какое-либо определенное значение, то в нем содержится то, что было запомнено ранее,— часть другой программы или какие-то «чужие» данные. Название «неопределенные» переменные в действительности означает, что значение переменных «непредсказуемо» или «лишено смысла». Некоторые компиляторы при обработке ссылок на переменные следят за тем, что-

бы при каждом таком обращении значение переменной уже было определено.

(г) Проверку на соответствие. Индексы массивов не должны выходить за пределы декларированных диапазонов. Фактические параметры должны быть того же типа и в том же количестве, что и формальные. Результаты арифметических вычислений не должны превышать размеров переменных, в которых они должны храниться. В некоторых компиляторах подобные ситуации контролируются и в случае ошибок программа будет завершена аварийно.

2. Пользуйтесь помощью компилятора.

Не пренебрегайте средствами отладки компилятора и операционной системы. Некоторые начинающие программисты считают для себя зазорным использовать имеющиеся в компиляторе средства отладки. Часто их рассуждения основаны на том, что проверка индексов увеличивает время работы программы (что имеет место), но во время первоначальных отладочных прогонов это ложная экономия. Небольшие затраты машинного времени, конечно, стоят меньше, чем значительные потери времени программиста и, возможно, еще бо́льшие потери времени у тех, кто ждет окончательной сдачи программы в работу.

Другая причина, вследствие которой не используют вспомогательные отладочные средства, основана на убеждении: «Я узнаю больше, если не буду пользоваться готовыми отладочными средствами». Если задача проработана тщательно, то программист хорошо представляет себе структуру программы и ход ее выполнения. Конечно, можно больше узнать из нового проекта, чем оставаясь в рамках старого. Поэтому, чтобы максимально повысить свою квалификацию и расширить кругозор, программист должен поработать над многими задачами, что можно сделать только при минимизации времени на отладку уже готовых программ. Это не следует интерпретировать как рекомендацию плохо или неполностью проводить отладку, а скорее как аргумент, подкрепляющий замечание о необходимости полного использования всевозможных отладочных средств.

3. Нужно предвидеть возможные ошибки.

(а) При планировании программы определите места, где могут возникнуть ошибки. Мы уже рекомендовали проводить проверку входных данных, поскольку это одно из основных мест в программе, которые доставляют неприятности. Теперь мы советуем, чтобы аналогично вы производили проверки в местах возможных ошибок внутри программы. Например, при выполнении операции деления A на B необходимо проверить, не имеет ли B нулевого значения. Если ваш компилятор не про-

веряет принадлежности индексов соответствующим диапазонам, то прежде, чем использовать величину 3*(IGRADE+1)/4+IPLUS, например, в качестве индекса, необходимо проверить, находится ли она внутри допустимых границ.

(б) Выделите ключевые точки внутри программы. Всякий раз, когда программа входит в новую фазу, полезно знать, что она выполнила одну группу инструкций и перешла к другой. Когда вы первый раз пишите программу, включите инструкции вывода в точки перехода от одной фазы к другой. В этом случае, если в программе произошла ошибка, вы будете иметь представление, в каком месте ее искать.

(в) При написании программы включайте отладочные инструкции. Постепенно вы привыкнете делать это автоматически даже тогда, когда ваш мозг полностью сосредоточен на самой программе. Например, в начале и в конце каждой подпрограммы следует автоматически ставить инструкции вывода.

(г) Избегайте многократного вставления и выбрасывания отладочных инструкций. Если вы нашли подходящие места и расположили там полезные для отладки инструкции, то оставьте их там вплоть до самых последних этапов работы над программой. Создается впечатление, что существует закон: когда вам кажется, что вы уже нашли и исправили последнюю ошибку, и вы удалили все фрагменты программы, помогавшие вам в отладке, появляется еще одна ошибка.

Вместо того чтобы удалять отладочные инструкции, можно просто их деактивировать. В качестве примера рассмотрим подпрограмму, вычисляющую темп обесценивания товара. Такая подпрограмма-функция может выглядеть следующим образом:

```
      REAL FUNCTION DDECLN (COST, NYEARS, YEARNO)
C  ВЫЧИСЛЕНИЕ ТЕМПА ОБЕСЦЕНИВАНИЯ
C  ВХОДНЫЕ ДАННЫЕ:
C  COST      НОВАЯ СТОИМОСТЬ ТОВАРА
C  NYEARS    КОЛИЧЕСТВО ЛЕТ, В ТЕЧЕНИЕ КОТОРЫХ ПАДАЛА СТОИ-
C            МОСТЬ ТОВАРА
C  YEARNO    ГОД, ДЛЯ КОТОРОГО НУЖНО ВЫЧИСЛИТЬ СНИЖЕНИЕ
C            СТОИМОСТИ (НАПРИМЕР, YEARNO =1 ОЗНАЧАЕТ ПЕРВЫЙ ГОД)
C  ВЫХОД:
C  DDECLN    ВЕЛИЧИНА СНИЖЕНИЯ СТОИМОСТИ В ЗАДАННОМ
C            ГОДУ
            INTEGER NYEARS, YEARNO
            REAL COST
            LOGICAL DEBUG/.TRUE./
            INTEGER  IDBOUT/6/
C             ОТЛАДОЧНЫЙ ВЫВОД ДЛЯ ТРАССИРОВКИ ПРОГРАММЫ
            IF(DEBUG) WRITE (IDBOUT, 100) COST, NYEARS, YEARNO
100         FORMAT ('*** ОТЛАДКА ** ПРИ ВХОДЕ В DDECLN
     X        GOST=',F10.2, 'NYEARS, YEARNO=',215')
                          ... ВЫЧИСЛЕНИЯ...
C             ОТЛАДОЧНЫЙ ВЫВОД ДЛЯ ТРАССИРОВКИ ВЫХОДА ИЗ ПРО-
C             ГРАММЫ
            IF(DEBUG) WRITE(IDBOUT, 101) DDECLN
101         FORMAT ('** ОТЛАДКА** ПРИ ВЫХОДЕ ИЗ DDECLN
     X        ВОЗВРАЩАЕТСЯ ЗНАЧЕНИЕ', F10.2)
            RETURN
            END
```

Когда программист первый раз прогоняет подпрограмму DDECLN, могут выявиться некоторые ошибки. Однако установить место их возникновения не представляет труда, так как при каждом обращении к подпрограмме все входные и выходные величины выводятся на печать. Поэтому, если подпрограмма постоянно дает ошибочные результаты, подозрение падает на некоторую общую часть подпрограммы. В том же случае, если только один набор данных дает неверные результаты, вы располагаете всеми значениями входных данных и можете проследить обработку этих значений в подпрограмме.

Важно отметить, что инструкции отладки *не* следует удалять после того, как программный модуль заработал. Вместо этого замените перфокарту

```
LOGICAL DEBUG /.TRUE./
```

на

```
LOGICAL DEBUG /.FALSE./
```

Это деактивирует отладочную печать. Однако, если в будущем понадобятся какие-либо изменения в модуле DDECLN, посредством одной перфокарты можно вновь изменить значение DEBUG на .TRUE. и активировать отладочные инструкции в указанном модуле. Спустя длительное время будет уже трудно вспомнить точки отладочной трассировки в программе, тогда как при разработке алгоритма определить их место сравнительно легко. Далее не имеет смысла терять время для подготовки инструкций вывода только для того, чтобы, выкинув их, вновь переписывать те же самые инструкции при последующих отладках. Выбросить всю отладочную информацию — это ложная экономия.

Можно обобщить этот метод. Один из возможных путей состоит в том, чтобы сделать DEBUG параметром или глобальной переменной (с помощью COMMON, как показано в разд. 2.6). Логическая переменная будет затем активизировать инструкции отладки во всей программе. Изложенный метод предполагает ввод специальных данных для управления переключателями отладки. Поэтому вместо того, чтобы вновь компилировать исходную программу после изменения инструкции, инициализирующей переменную DEBUG, вы можете включать или выключать отладочные инструкции посредством специальных значений во входных данных. В этом случае, если при обработке одного набора входных данных происходят какие-либо ошибки, можно активировать отладку только для этого набора входных данных, тем самым ограничивая вывод на печать только критическими участками программы.

Другой вариант этой схемы предполагает назначение «уровня доверия» для каждого участка отлаживаемой программы. Так, простая программа, связанная с выполнением элементарных вы-

числений, может иметь высокий уровень доверия, тогда как сложная подпрограмма, выполняюшая много запутанных вычислений, может иметь низкий уровень доверия. DEBUG задают в этом случае в виде числовой переменной. При выполнении программы, если значение DEBUG не выше уровня доверия некоторой программы или инструкции вывода, то отладочный вывод не происходит. Однако, если значение DEBUG выше уровня доверия определенной секции программы, происходит отладочный вывод для этой секции. Локальные инструкции отладки могут быть заданы в виде

```
IF (DEBUG.GT.4) WRITE(—,—) MAXVAL, COST
```

Процесс отладки начинается при высоком значении DEBUG; при этом происходит вывод на печать из всех отладочных точек программы. Когда отладка программы продвинулась и вы добились большего доверия в программных модулях, уменьшите значение DEBUG. Нулевое значение DEBUG будет указывать, что отладочная печать вообще не нужна.

Один из наиболее часто выдвигаемых аргументов против того, чтобы оставлять в программе отладочные инструкции, заключается в том, что они требуют дополнительных затрат времени, памяти, бумаги. Этот аргумент часто выдвигают опытные программисты, ранее программировавшие на старых машинах. Машины работали тогда медленнее и были меньше многих современных ЭВМ. Для относительно сложных программ время проверки значения одной логической переменной или требуемая емкость памяти для конструкции вида IF (— —) WRITE (—, —) — — — незначительны. Следует, конечно, разумно подходить к использованию инструкций отладки, располагая их по возможности таким образом, чтобы они повторно не исполнялись. (Внутрь цикла DO, например который будет повторен 1000 раз, не следует вводить инструкцию отладки, если вам не требуется управлять каждой итерацией цикла. Вы можете, однако, использовать «выборочный» метод, выводя значения на 10, 20, 30-й и т. д. итерациях.) Если выбирать между эффективным выполнением программы и легкой отладкой, то в производственных программах, которые будут использованы многократно или с большими объемами данных, предпочтение отдают обычно эффективности. Однако даже и в них следует по возможности оставлять отладочные инструкции. Мы рассмотрим вопрос об эффективности программы более детально в разд. 1.6.

(д) Легче отлаживать правильный алгоритм, чем неправильный. Время, затрачиваемое на то, чтобы сделать алгоритм программы еще до того, как она будет запрограммирована, правильным, отшлифованным и проверенным, считается хорошо потраченным временем. Не стоит слишком беспокоиться о написании

программы. Гораздо легче переделать или даже вновь написать программу, чем заново переработать ее алгоритм.

(е) Полезно объяснить детально алгоритм еще кому-нибудь. В процессе обсуждения вы сможете найти моменты, которыми ранее пренебрегли. С другой стороны, тот, кому вы объясняете алгоритм, может высказать мнение, что тот или иной сегмент «выглядит сомнительно». Часто от собеседника требуется быть просто слушателем; вы сами найдете ошибки, когда попытаетесь объяснить алгоритм. Многие коммерческие группы программирования используют прием, который называется — «прогулкой по программе» [1]. Для этого каждый член группы объясняет свои разработки остальным сотрудникам группы. Руководитель обычно не присутствует при таких обсуждениях, и поэтому члены группы не чувствуют себя как на суде. Обсуждающие вносят предложения, но чаще всего сама попытка объяснить логику программы помогает автору выявить потенциальные затруднения. Ясно, что этим методом могут также пользоваться студенты, отлаживающие программы; все, что требуется, это другие студенты, которые согласились бы послушать.

1.5.2. Как же все-таки производят отладку?

Возможно, вас удивило, когда вы написали первые программы и отдали их более опытному программисту, что он смог быстро найти и исправить ваши ошибки. Опытный программист смог быстро найти ошибки потому, что хорошо знает язык, понимает, где могут возникнуть ошибки в ваших относительно простых первых программах. В своей практике он встречался со многими различного рода ошибками. С приобретением опыта в программировании скорость, с которой вы будете производить отладку программы, также увеличится. Невозможно детализировать все шаги отладки конкретного задания или объяснить все возможные методы отладки. Некоторые лучшие приемы отладки заключаются в комбинировании, кажется, не связанных друг с другом моментов и формировании гипотез. И все же мы рассмотрим некоторые конкретные приемы отладки, применяемые опытными программистами.

Сначала вы должны научиться понимать диагностические сообщения, выдаваемые компилятором. К сожалению, некоторые компиляторы выдают одно и то же туманное сообщение («SYNTAX ERROR», т. е. синтаксическая ошибка) для всех ошибочно записанных инструкций. В этом случае вы должны проверить син-

---

[1]) Этот прием называют также структурным разбором (см. Э. Йодан). Структурное проектирование и конструирование программ.— М.: Мир, 1979) или структурным рецензированием.— *Прим. ред.*

таксис инструкции с помощью руководства по языку, обращая внимание на парность скобок, посмотреть, не пропущена ли запятая, нет ли ненужных знаков и т. д.

Другие, более «развитые» компиляторы выдают детальные сообщения, основанные на предположении о том, какая инструкция должна быть на самом деле. Однако иногда такие компиляторы могут ошибаться. Приведем классический пример из Фортрана. В Фортране может быть только 2 случая, когда терм арифметического выражения может состоять из имени, после которого непосредственно следует левая скобка: при ссылке на массив или на функцию. Используя метод исключения, если компилятор не встретил декларации массива с этим именем, он заключает, что имя, за которым следует левая скобка, является обращением к функции. Если вы просто забыли объявить массив или допустили орфографическую ошибку в имени массива, компилятор выдает неправильные сообщения о том, что якобы имеют место ссылки на несуществующие функции. Только опыт помогает разобраться в таких случаях. Поэтому вам следует хорошо знать сообщения об ошибках. Всякий раз, когда вы встречаетесь с новыми сообщениями об ошибках или с новыми ситуациями, в которых происходят сообщения об ошибках, следует взять на заметку такие обстоятельства и сопутствующие им сообщения. Еще более гибкие и развитые компиляторы могут вносить исправления в программу. Приведем пример. Элементы в списке инструкций FORMAT должны быть разделены запятыми. В некоторых случаях, таких, как пропуск запятой после завершающей кавычки в цепочке литер, вполне очевидно, где должна стоять запятая. В случае ошибки вида

```
12    FORMAT ('ANS='10X, A5)
```

компилятор может не только указать, что нужно поставить запятую между 'ANS=' и 10X, но и вставить ее, сообщив вам об исправлении, и продолжить компилирование «исправленной» программы. Разработчик компилятора, анализируя ситуации, в которых кавычка могла предшествовать цифре, установил, что в большинстве, скажем в 80%, случаев между кавычкой и цифрой подразумевалась запятая. Это хорошо для большинства случаев, но что, если ваша конкретная ошибка попадает в категорию меньшинства.

В этом случае такая «коррекция» может увести гораздо дальше от правильной инструкции, чем та, с которой вы начинали. Поэтому, если компилятор выдает, например, сообщение «пропущена запятая» или исправляет инструкцию путем занесения запятой, еще раз проверьте синтаксис инструкции. Вы, а не компилятор написали исходную инструкцию и не компилятор, а вы знаете, что в данном случае подразумевается.

После того как вы справились с сообщениями об ошибках, выданных компилятором, и располагаете программой, которая правильно скомпилирована, вы подготовлены к тому, чтобы запустить программу на выполнение. К этому моменту у вас должен быть один чистый листинг исходной программы, в который включены вся документация, описывающая программу, имена и назначение основных переменных, в программу должны быть «включены» все необходимые отладочные инструкции, вы должны далее получить распечатку используемых входных данных и только после этого можно начать выполнение программы.

Для отладки программ не было найдено пока ничего лучше, чем «ручное» моделирование процесса выполнения программы. При моделировании «вручную» вы держите в руке карандаш и начинаете «играть роль машины»; вы записываете значения переменных и прослеживаете за ходом работы программы. При этом вы сравниваете каждое полученное при трассировке значение со значением, которое та или иная переменная должна иметь на соответствующем шаге алгоритма.

При моделировании «вручную» вам может помочь трассировка хода выполнения программы, создаваемая или компилятором, или вашими собственными отладочными инструкциями. Если вы сами строите трассировку хода выполнения программы, потребуется большое число сообщений типа «это место пройдено». Такие простые сообщения делаются всякий раз после того, как выполнение программы переходит от одного шага алгоритма к другому. Например: «СОРТИРОВКА ОКОНЧЕНА; НАЧАЛОСЬ ВЫЧИСЛЕНИЕ СРЕДНЕГО ЗНАЧЕНИЯ». (Требуется несколько лишних ударов по клавишам устройства набивки данных, чтобы сообщения стали гораздо более осмысленными, чем, например, сообщения типа «ТОЧКА 1». Конечно, это с лихвой окупается во время отладки.) Вход в подпрограмму и выход из нее представляют собой две естественные точки для трассировки программы, поскольку эти точки связаны, как правило, с переходными шагами алгоритма.

Чтобы найти ошибку, часто достаточно построить схему последовательности прохождения алгоритма. Бывают, конечно, случаи, когда выдерживается правильный порядок выполнения инструкций в программе, но при этом получаются неверные результаты. Тогда необходимо более глубокое моделирование; вам придется проследить также за вычислением значений по ходу выполнения программы.

Для всех переменных нужно произвести вычисления точно так же, как это должно происходить в программе. Если ошибка невелика, то все вычисления необходимо производить с полной точностью. (Не пренебрегайте также ошибками округления и усечения. Эти ошибки обсуждаются в гл. 6.) Если ошибка велика,

то обычно достаточно лишь оценить значения на каждом шаге вычислений, запоминая моменты, когда оценка или слишком велика, или слишком мала и насколько существенно она расходится с результатами счета.

Во время этой фазы отладки используют распечатки значений, которые получаются по ходу вычислений. В этом случае распечатка подобна той, которая выдается при трассировке программы, только вместо сообщений типа «это место пройдено» появляется «вычислено значение X, оно равно — — — —». Ключевые точки при выполнении программы становятся также ключевыми точками для вычисляемых величин. Как и ранее, вывод при трассировке должен быть четко помечен, чтобы определить 1) местонахождение инструкций вывода в программе, 2) знать, какие переменные прослеживаются и 3) какие они имеют значения.

Подходите разумно к выполнению этой фазы отладки. Часто нет необходимости просматривать значения всех элементов массива. Например, если производится сортировка массива, достаточно значений первых десяти элементов для того, чтобы убедиться, что сортировка работает. Если элементы последовательно добавляются к массиву, то можно удовлетвориться значениями последней полудюжины элементов. Выводите на печать столько сведений, сколько необходимо для наблюдения за выполнением программы, но не настолько много, чтобы утонуть в бесполезной информации.

Другой метод отладки, который может быть очень полезным, состоит в прослеживании работы программы в обратном направлении. Предположим, что вы находитесь на 200-й инструкции и значение X неверно. От этой точки вы просматриваете программу вверх и ищете место, где X было присвоено это значение. Разумеется, поиск не сводится к простому продвижению вверх по программе, поскольку инструкции GO TO или IF могли изменить последовательность выполнения других инструкций. Если X в некоторой точке присваивается значение Y+Z, то, возможно, или Y, или Z имели ошибочные значения. Тогда придется проследить, как Y и Z получили свои значения. Таким образом строится дерево, где каждый вариант, в котором X может получить то или иное значение, представлен единственной ветвью. Заметим, что, если X зависит от Y и Z, одна из ветвей X в этом дереве может расщепиться на подветви Y и Z. Каждую из двух новых ветвей прослеживают в обратном направлении по отношению к последовательности выполнения программы. Эти ветви затем удаляют одну за другой, так как 1) моделирование и вывод программы подтверждают, что вычисления в данной ветви и все ее входные данные правильны, или 2) моделирование и вывод программы подтверждают, что данная ветвь вообще не выполня-

лась. В оставшиеся ветви можно ввести дополнительные инструкции отладки, чтобы сузить поиск ошибочных вычислений, точно определяя, какая ветвь исполнялась и какие начальные и конечные значения имели все переменные на этой ветви. В конечном счете этот метод приведет к точке возникновения ошибки. Лишь немногие компиляторы дают возможность выполнить процесс обратного прослеживания за ходом выполнения программы.

Вероятно, труднее всего при отладке прийти к заключению, что ошибок больше чем одна. Следует проанализировать весь напечатанный материал и убедиться, что вы нашли столько ошибок, сколько можно выявить по имеющемуся выводу программы. Многие программисты практикуют «итеративную отладку», при которой они получают распечатку, находят одну ошибку, исправляют ее, вновь выполняют программу, находят следующую ошибку, исправляют ее, снова дают вывод на печать, находят следующую ошибку и т. д. Ваше время будет использовано более эффективно, если вы продолжите моделирование вручную до тех пор, пока не проследите значительную часть программы, несмотря на то, нашли ли вы одну или 100 ошибок. Если вы уже настроились на моделирование, то легче его продолжить, чем прервать, а затем вновь возвращаться к моделированию после следующего выполнения программы. Заманчиво исправить «последнюю ошибку», но опытные программисты ясно понимают, что «последняя ошибка» часто оказывается далеко не последней.

1.6. Эффективность

Эффективными называют программы, которые используют минимальное количество ресурсов. Важно знать, что существует много различных видов ресурсов: машинное время и память — два важных ресурса, вспомогательная внешняя память (магнитные ленты и диски; см. гл. 4) и бумага также относятся к ресурсам ЭВМ. Два очень ценных ресурса, которым часто не придают значения, это время работы программиста и время пользователя. Когда мы используем термин «эффективность», необходимо тщательно определить, какие ресурсы экономятся и насколько эта экономия увеличивает расход других ресурсов. Относительная ценность указанных ресурсов существенно различна для различных ЭВМ и различных людей. Важно, однако, поставить в соответствие каждому ресурсу какую-то разумную стоимость.

1.6.1. Эффективность в смысле машинного времени/памяти

Эффективность в смысле машинного времени означает потребление наименьшего количества времени для получения желаемого результата. Ясно, что следует избегать ненужных вычисле-

ний. Ниже приведены классические случаи, очевидно, неэффективных программных фрагментов.

Один из наиболее серьезных видов неэффективности — это неоднократное повторение вычислений одних и тех же значений. Следует выносить постоянно повторяющиеся вычисления из цикла. Рассмотрим пример:

```
      DO 10 I = 1,N
         B(I) = A(I)/(C**2 - 4*D)
   10 CONTINUE
```

Очевидно, что (C**2—4*D) не изменяется внутри цикла. Однако большинство компиляторов строят этот цикл в объектной программе так, что при каждом проходе через цикл происходит вычисление (C**2—4*D). Наиболее простое решение — вынести инвариантные вычисления за пределы цикла:

```
      DENOM = (C**2 - 4*D)
      DO 10 I = 1, N
         B(I) = A(I) / DENOM
   10 CONTINUE
```

Случаи, подобные рассмотренному, встречаются довольно часто.

Второй очевидный источник неэффективности — одновременное применение вещественных и целых чисел и переводы одних в другие. На многих ЭВМ вычисления с вещественными числами требуют значительно больше времени, чем с целыми; например, на малых ЭВМ выполнение арифметической операции с двумя вещественными числами требует обращения к подпрограмме. Преобразования из одного типа данных в другой также вызывают расход машинного времени. Поэтому всегда, когда есть выбор между вещественными и целыми числами, следует отдавать предпочтение вычислениям с целыми числами. Аналогично следует избегать ненужных преобразований между типами. Там, где это возможно, надо стараться, чтобы выражения смешанного типа содержали только один член «инородного» типа, и его следует расположить в таком месте, чтобы с ним было связано по возможности наименьшее количество преобразований.

Третий очевидный фактор, влияющий на временные затраты,— способы индексации. Напомним из гл. 0, что формула доступа к индексированной переменной включает по одному дополнительному члену на каждое лишнее измерение, и чем больше размерность, тем больше умножений приходится на каждый член. По этой причине ссылки на одномерные массивы предпочтительнее, чем на двумерные и т. п. Заметим, однако, что если индекс представляет собой сложное выражение типа (3*I*J+7*I—J), то использовать его при ссылках, наверное, хуже, чем два простых

индекса (I, J). Очевидно, что к простым переменным обращаться намного легче, поэтому, где это возможно, их следует предпочесть массивам.

Однако несправедливо игнорировать в этом процессе роль компилятора. Сделанные выше замечания относятся к общеизвестным методам обеспечения эффективности. Точно так же, как программисты в течение длительного времени при написании программ учитывают изложенные соображения эффективности, так и разработчики компиляторов добиваются эффективности. Даже многие компиляторы прикладного характера обнаруживают сегодня отмеченные выше неэффективные конструкции и генерируют эффективные программы. Большинство источников неэффективности программ было связано с компиляцией инструкций Фортрана в объектный код. Возьмем, к примеру, повторяемые вычисления внутри цикла. Часто такие ситуации возникают в генерируемом компилятором коде для вычисления адресов элементов массива или для определения адресов вызова подпрограммы. Эти части кода программы не поддаются прямому управлению со стороны программиста, и поэтому компилятор должен сам оптимизировать такого вида ситуации. Многие компиляторы выносят за пределы цикла инвариантные вычисления вне зависимости от того, программист ли оставил эти вычисления внутри цикла, или они возникли в процессе компиляции.

Мы рекомендуем вам не пользоваться заведомо неэффективными конструкциями. Однако не стоит и корпеть над инструкциями программы слишком долго ради микросекундного выигрыша; это не есть показатель квалификации программиста. В конечном итоге разница во времени выполнения между хорошо составленной программой и наилучшей программой, реализующей тот же алгоритм, оказывается несущественной. Важнее то, что, когда программист прибегает к разного рода хитростям, стремясь во что бы то ни стало повысить эффективность выполнения, наглядность текста программы часто ухудшается; на наш взгляд, это серьезное нарушение приоритета. Короче, мы рекомендуем, конечно, проявлять здоровую заботу об эффективности программы, но не одержимость.

1.6.2. Эффективность алгоритма

Мы сделали довольно беглый обзор по вопросам эффективности нижнего уровня, беспокоясь в первую очередь об оптимальном программировании конкретных шагов алгоритма. Гораздо более существенной оказывается, однако, сложность самого алгоритма. Давайте рассмотрим два примера — алгоритмы поиска и сортировки.

Существует много методов поиска определенного значения в списке значений. Наиболее прямолинейный из них — метод последовательного перебора. Значение, которое вы ищите в списке, сравнивается с каждым элементом списка, начиная с первого элемента, до тех пор, пока значения не совпадут. Ясно, что в среднем потребуется просмотреть половину списка, чтобы найти нужный элемент. Если список состоит из N элементов, в среднем понадобится N/2 сравнений.

Рассмотрим теперь время работы программы, реализующей этот алгоритм. Для списка из N элементов будет в среднем выполнено N/2 сравнений плюс N/2 приращений индекса цикла и проверок (для выхода из цикла), плюс некоторое постоянное количество других выполняемых инструкций. Таким образом, время выполнения данного алгоритма составляет $c*(N/2)+d$, где c и d — константы, зависящие от времени выполнения операций сложения или сравнения на конкретной ЭВМ. Ясно, что этот алгоритм выполняется со скоростью, пропорциональной N — количеству элементов массива. Говорят, что алгоритм имеет *временну́ю сложность N*.

Однако это не самый эффективный метод поиска. При поиске нужного номера в телефонном справочнике мы не просматриваем подряд все номера. Более распространенный метод поиска заключается в последовательном приближении: после каждой неудачной попытки мы продвигаемся вперед, если предположение дало слишком «маленький» результат, и назад, если слишком «большой». Когда поиск подходит ближе к желаемому месту в книге или таблице, дистанция между последовательными попытками становится существенно меньше. Можно аппроксимировать этот подход посредством метода, известного под названием «бинарный поиск». В этом методе изначальная попытка происходит в середине таблицы. Если искомое значение «меньше» или «больше» элемента из середины таблицы, то поиск продолжается в первой или во второй половине таблицы соответственно. Следующая попытка относится к середине выбранной половины (т. е. 1/4 или 3/4 таблицы). Затем мы продолжаем поиск в верхней или нижней части этих четвертей в зависимости от того, меньше или больше искомое значение очередного пробного значения из таблицы. Процесс последовательного деления таблицы пополам продолжается до тех пор, пока или не будет найдено желаемое значение, или размер таблицы не уменьшится до нуля.

Метод бинарного поиска иллюстрирует подпрограмма на рис. 1.2.

Время выполнения бинарного поиска значительно меньше, чем простого последовательного поиска. Каждый раз, когда мы делим список, мы отбрасываем его половину. На первом шаге список уменьшается на половину предыдущего размера, на вто-

```
      SUBROUTINE BISRCH (ARG, TABLE, TABSIZ, LOCN)
C
C     ПОДПРОГРАММА БИНАРНОГО ПОИСКА
C
C     ПОИСК В ТАБЛИЦЕ "TABLE" ЗНАЧЕНИЯ "ARG" МЕТОДОМ БИНАРНОГО
C     ПОИСКА. ПРОЦЕСС ПОСЛЕДОВАТЕЛЬНОГО ДЕЛЕНИЯ ПОПОЛАМ ТАБЛИЦЫ
C     ПРОДОЛЖАЕТСЯ ДО ТЕХ ПОР, ПОКА НЕ БУДЕТ НАЙДЕН ИСКОМЫЙ
C     ЭЛЕМЕНТ ИЛИ ПОКА НЕ БУДЕТ ИСЧЕРПАНА ТАБЛИЦА.
C
C     ВХОДНЫЕ ПАРАМЕТРЫ:
C     ARG       ЗНАЧЕНИЕ, КОТОРОЕ ИЩЕТСЯ В ТАБЛИЦЕ;
C     TABLE     ТАБЛИЦА — СПИСОК, В КОТОРОЙ ИЩЕТСЯ ЗНАЧЕНИЕ ARG;
C     TABSIZ    РАЗМЕР ТАБЛИЦЫ;
C     ВЫХОДНЫЕ ПАРАМЕТРЫ:
C     LOCN      ЯЧЕЙКА, В КОТОРОЙ БЫЛ НАЙДЕН "ARG" ИЛИ 0, ЕСЛИ "ARG"
C               НЕ БЫЛ НАЙДЕН.
C
C     ВНУТРЕННИЕ ПЕРЕМЕННЫЕ:
C     LOW       НАИМЕНЬШИЙ ИНДЕКС АКТИВНОЙ ЧАСТИ ТАБЛИЦЫ;
C     HI        НАИБОЛЬШИЙ ИНДЕКС АКТИВНОЙ ЧАСТИ ТАБЛИЦЫ;
C     MID       СРЕДНЯЯ ТОЧКА АКТИВНОЙ ТАБЛИЦЫ;
C     SIZE      РАЗМЕР АКТИВНОЙ ТАБЛИЦЫ.
C
C     ПРОГРАММИСТ: — — —          ДАТА: — — —


      INTEGER ARG, LOCN, TABSIZ, TABLE(TABSIZ)
      INTEGER LOW, HI, MID, SIZE
      INTEGER IDBOUT
      LOGICAL DEBUG
      DATA DEBUG/.FALSE./,IDBOUT/6/
      IF (DEBUG) WRITE (IDBOUT,11)
   11 FORMAT (' ENTRY TO BISRCH')
C
C        НАЧАЛЬНАЯ ПОПЫТКА НА ВСЕЙ ТАБЛИЦЕ ОТ 1 ДО TABSIZ
      LOW = 1
      HI = TABSIZ+1
      SIZE = TABSIZ
C        ВНАЧАЛЕ LOCN: „НЕ НАЙДЕНО"; ИЗМЕНИМ, КОГДА „ARG" БУДЕТ НАЙДЕН
      LOCN = 0
C
C        *** ОСНОВНОЙ ЦИКЛ ***
C
C        ТАБЛИЦА ИСЧЕРПАНА?
   25 IF (SIZE .LE. 0) GO TO 150
C        ДЕЛИМ ТАБЛИЦУ ПОПОЛАМ; ВЫЧИСЛЯЕМ СРЕДНЮЮ ТОЧКУ
      SIZE = SIZE/2
      MID = (LOW+HI)/2
      IF (DEBUG) WRITE (IDBOUT,35) LOW, HI, ARG, MID, TABLE(MID)
   35 FORMAT (' BISRCH LOOP: LOW, HI BOUNDS=',2I10, '; COMPARING',I10,
     X   ' TO TABLE(',I6,') =',I10)
      IF (ARG .LT. TABLE(MID)) HI = MID
      IF (ARG .GT. TABLE(MID)) LOW = MID
      IF (ARG .NE. TABLE(MID)) GO TO 25
C
C        ИСКОМЫЙ ЭЛЕМЕНТ НАЙДЕН
      LOCN = MID
C
C        ОБЩАЯ ТОЧКА ВОЗВРАТА
  150 IF (DEBUG) WRITE (IDBOUT,151) ARG, LOCN
  151 FORMAT (' BISRCH EXIT: ARG=',I10,',  LOCN=',I10)
      RETURN
      END
```

Рис.1.2. Подпрограмма банарного поиска. (11: «вход в BISRCH»; 35: «цикл BISRCH: границы LOW,HI=», «сравниваются»; 151: «выход из BISRCH:»)

ром шаге — на одну четверть, на третьем шаге — еще на одну восьмую и т. д.; за k шагов список сокращается до $1/(2**k)$ от первоначального размера. Если размер списка равен N и если $(2**k) > N$, то список исчерпан и работа окончена. Таким образом, максимально возможное количество шагов поиска соответствует такому наименьшему значению k, что $(2**k) > N$. Это значение k равно $\log_2(N)+1$. Поэтому в наихудшем случае мы сделаем не более $\log_2(N)$ сравнений до того, как список будет исчерпан. Временна́я сложность этого процесса не может быть хуже $\log_2(N)$. При больших значениях N этот метод дает существенное преимущество по сравнению с последовательным просмотром. Однако метод бинарного поиска применим только к отсортированным таблицам, что существенно ограничивает сферу применения этого метода.

В программе было проверено количество сравнений, необходимых для того, чтобы найти заданное значение в отсортированной таблице из 100 элементов, каждый из которых имеет значение в диапазоне от 1 до 1000. Было выбрано в таблице 10 случайных элементов при использовании обоих методов — последовательного и бинарного поиска. Количество сравнений, необходимых для каждого из указанных методов, показано в табл. 1.1.

Таблица 1.1. *Сравнение бинарного и последовательного поиска*

ТЕСТ №	ИСКОМОЕ ЗНАЧЕНИЕ	КОЛИЧЕСТВО СРАВНЕНИЙ ПОСЛЕДОВАТЕЛЬНЫЙ	БИНАРНЫЙ
1	418	51	1
2	946	95	6
3	477	56	7
4	3	1	7
5	725	74	6
6	326	38	3
7	432	53	7
8	656	70	6
9	51	5	6
10	401	49	6
В СРЕДНЕМ		49.20	5.50

Таблица подтверждает установленную ранее статистику временно́й сложности: 100/2 равно 50, а $\log_2(100)$ находится между 6 и 7. Преимущество бинарного поиска еще более заметно, если размер таблицы растет; 1000/2 дает 500, в то время как $\log_2(1000)$ — менее 10. Различие между N/2 и $\log_2(N)$ увеличивается с ростом N.

Давайте теперь проанализируем другой алгоритм — сортировку. Под сортировкой мы подразумеваем расположение серии значений в таком порядке, когда каждое данное значение мень-

ше[1] последующего. Предположим, что у нас есть N элементов от A(1) по A(N).

Классический «метод пузырька» включает сравнение последовательных пар чисел и обмен их местами, если числа расположены не по порядку, т. е. когда большее число предшествует меньшему. Эта процедура последовательно продолжается с массивом чисел, пока очередной проход не завершится без обмена. Алгоритм этого процесса может быть представлен следующим образом:

1. M=N—1.
2. Если M < 2, останов.
3. Переключатель=«выключен».
4. Выполнить для I=1 до M:
   если A(I) и A(I+1) расположены не по порядку (A(I) > >A(I+1)), то поменять их местами и установить переключатель в положение «включен».
5. M=M—1.
6. Если переключатель выключен, то останов; в противном случае вернуться на шаг 2.

Определим временну́ю сложность этого алгоритма. Цикл на шаге 4 включает наибольшее количество действий; поэтому мы вначале оценим, каково число повторений в цикле. Так как M на шаге 5 уменьшается на 1, то число повторений инструкций, выполняемых на шаге 4, уменьшается на 1. Первый раз происходит N — 1 сравнений, во второй раз N — 2 и т. д., и в последний раз — одно сравнение. Среднее число сравнений, выполняемых на шаге 4, равно N/2.

Хотелось бы узнать, сколько раз на шаге 6 переключатель будет установлен в положении «включен». Переключатель находится в положении «включен» только в случае, если два соседних элемента на шаге 4 поменялись местами. Заметим, что сразу после обмена мы переходим к следующему элементу. Поэтому за один проход меньший из сравниваемых элементов сдвигается только на одну позицию вверх. Однако наибольший элемент за один проход перемещается в конец списка. При втором проходе второй наибольший элемент занимает свое место. Поэтому достаточно (N—1) проходов для размещения всех элементов на свои места. Хотя количество проходов для сортировки зависит от начального распределения данных, в среднем требуется N/2 проходов для сортировки N чисел.

[1]) Мы будем говорить «меньше» только для простоты. Однако должно быть ясно, что те же методы применимы и в том случае, когда значения дублируются (тогда «меньше или равно» было бы точнее) или когда сортировку следует выполнять в порядке уменьшения значений (здесь правильнее сказать «больше чем»).

Так как шаг 4 включает в среднем N/2 сравнений на проход и в среднем выполняется N/2 проходов, то количество сравнений, необходимых для проведения всей сортировки, пропорционально N**2. Поэтому говорят, что данный алгоритм порядка N**2. Это значит, что если мы увеличиваем размер сортируемого массива в 10 раз, то следует ожидать увеличения времени выполнения сортировки в 100, или 10**2, раз. Поэтому нужно дважды подумать, прежде чем использовать этот алгоритм для сортировки 1000 чисел, так как в этом случае понадобилось бы в среднем миллион сравнений.

Ни один из кратко рассмотренных выше методов повышения эффективности, никакие хитрости или тщательное кодирование программы не улучшают этой ситуации, так как это неотъемлемое свойство алгоритма N**2. Сложность N**2 характеризует сам *алгоритм*, а не способ *кодирования* этого алгоритма. Если мы хотим решить проблему сортировки эффективнее, необходимо искать более быстрые алгоритмы, а не заниматься различными методами программирования заведомо медленного алгоритма.

Лучшим алгоритмом является алгоритм быстрой сортировки по Хоору. В этой сортировке не происходит обмена местами между соседними элементами, так как в этом и заключается причина неэффективности метода «пузырька». Здесь устанавливается точка «деления» так, чтобы все элементы по одну сторону от элемента, находящегося в точке деления, были меньше этого элемента, а все элементы по другую сторону — больше его. (Заметим, что элементы на любой стороне могут не быть расположены в правильном возрастающем порядке; однако все элементы слева будут меньше, а все элементы справа — больше разделительного элемента.) Это приводит к образованию двух списков, по одному с каждой стороны от точки деления. Полученные два списка можно сортировать независимо, так как все элементы одного списка будут больше элементов другого списка. Кроме того, каждый из образованных списков будет меньше начального списка.

Выберем один из крайних элементов в качестве первого разделительного элемента; предположим, что мы сделали левый элемент разделительным. Сортировку начнем с правого конца списка, сравнивая все элементы по отношению к разделительному и оставляя на месте те элементы, которые больше разделительного. Мы пытаемся определить место точки деления списка, и, если анализируемые элементы больше разделительного элемента, они будут оставаться справа от него. Однако, когда будет найден элемент, который меньше разделительного, он должен быть перенесен по другую сторону списка. Проделаем это, поменяв местами разделительный и найденный меньший элемент. Теперь продолжим тот же самый процесс слева направо, т. е. будем остав-

лять на месте элементы, если они меньше разделительного. Когда будет найден больший элемент, мы поменяем его местами с разделительным элементом. Заметим, что всякий раз, когда мы меняем местами элементы, меньший элемент перемещается влево от разделительного или больший элемент перемещается вправо. Продолжаем этот процесс до тех пор, пока все элементы не распределим на два списка: элементы меньше разделительного элемента и элементы, большие его.

Вначале проиллюстрируем метод примером. Используем два указателя L и R (левый и правый) для пометки текущего сравниваемого элемента с разделительным.

1.	45	32	62	22		18	75	57
	L					x	←	R
2.	18	32	62	22		45	75	57
	L	→	x			R		
3.	18	32	45	22		62	75	57
			L	x	←	R		
4.	18	32	22	45		62	75	57
			L	→ R				
5.	18	32	22	45		62	75	57
				Div				

На первом шаге этого процесса элемент (57), идентифицированный правым (R) указателем, сравнивается с элементом (45), помеченным левым (L) указателем. Пока левый элемент меньше правого элемента, мы перемещаем правый указатель влево. Однако, когда левый элемент оказывается больше правого элемента, меняем их местами (18 и 45 в строке 2). Затем начинаем двигать левый указатель до тех пор, пока правый элемент (45) не окажется меньше левого элемента (62), и вновь меняем элементы местами (строка 3). Продолжаем процесс перемещения правого и левого указателя и обмен элементов местами до тех пор, пока указатели не встретятся. Элемент, на котором встретились указатели, и будет разделительным элементом (Div).

Затем повторим весь этот процесс применительно к каждому из двух списков по обе стороны от точки деления. Если каждый из двух списков содержит нули или единицы, то список уже отсортирован. Условие окончания соблюдается только тогда, когда все списки отсортированы. Алгоритм быстрой сортировки приведен на рис. 1.3.

Полный анализ времени выполнения алгоритма быстрой сортировки не входит в рассмотрение данной книги. Кнут установил, что количество сравнений, требуемых для этого алгоритма,

пропорционально $N*\log_2(N)$, где N — размер сортируемого массива. Чтобы убедить вас в какой-то степени, что эта оценка верна, выясним, чему равно количество сравнений, выполняемых в лучшем случае. Вначале мы начнем работу с одним списком размера N. Этот список разделится на два списка, в лучшем случае

```
Сортировка массива A размера N в порядке возрастания
1.  LIST-TO-SORT = (1,N)
2.  DO STEPS 3—7 WHILE LIST-TO-SORT не пустой
3.  Изымаем последнюю пару из стека LIST-TO-SORT. Пусть LEND —
    первый, а REND — второй элемент этой пары.
    LPT = LEND; RPT = REND
4.  DO STEPS 5—6 WHILE LPT < RPT
5.      ** Вначале перемещаем левый указатель, насколько возможно **
        DO WHILE (A(LPT) ≤ A(RPT))
            LPT = LPT + 1
            IF(LPT ≥ RPT) GO TO STEP 7
        END
        ** Найден элемент, расположенный не по порядку            **
        ** Пересылаем левый элемент в правую часть                 **
        Меняем местами A(LPT) и A(RPT)
6.      ** Теперь перемещаем правый указатель, насколько возможно **
        DO WHILE (A(LPT)) ≤ A(RPT))
            RPT = RPT — 1
            IF(LPT ≥ RPT) GO TO STEP 7
        END
        ** RPT элемент переходит на другую сторону                **
        Меняем местами A(LPT) н A(RPT)
7.      ** RPT — точка деления                                    **
        ** Делим список на два и сортируем каждый независимо       **
        ** Избегаем сортировки пустых списков                      **
        IF(RPT — 1 > LEND) поместить (LEND, RPT — 1) в LIST-TO-SORT
        IF(RPT + 1 < REND) поместить (RPT + 1, REND) в LISTD-TO-SORT
        END
```

Рис. 1.3. Алгоритм быстрой сортировки.

размер каждого из которых равен N/2. (Так как начальный крайний правый элемент становится точкой деления, точный размер каждого нового списка зависит от того, насколько хорошо был сбалансирован список по отношению к этой точке деления.) Каждый из двух новых списков делится на два списка, что дает 4 списка размерами N/4, которые в свою очередь дадут 8 списков размерами N/8 и т. д. В лучшем случае этот процесс оканчивается $2**k$ списками размером $N/(2**k)$ для наибольшего значения k, при котором $(2**k) \leqslant N$. Данное значение k точно равно $\log_2(N)$. Процесс деления на два всех подсписков должен быть выполнен $\log_2(N)$ раз. Заметим, что деление списка требует анализа всех элементов списка и что общее количество элементов всех списков равно примерно N (за вычетом точек деления). Поэтому каждый из указанных $\log_2(N)$ проходов включает примерно N сравнений.

Следовательно, количество сравнений, выполняемых алгоритмом быстрой сортировки, не может быть меньше $N*\log_2(N)$.

Для всех N значение $\log_2(N)$ меньше, чем N, и, когда N растет, $\log_2(N)$ также растет, но намного медленнее. Например, $\log_2(10)$ равен примерно 3, а $\log_2(1000)$ несколько меньше 10. Поэтому 1 000 000 сравнений, необходимых для сортировки 1000 элементов по методу «пузырька», можно снизить до 10 000 сравнений при использовании метода быстрой сортировки. Понятно, что метод быстрой сортировки дает существенную экономию. Совершенно очевидны преимущества выбора хорошего алгоритма и реализации его простым и наглядным образом.

1.7. Изменяемость программы

Практически все «производственные» программы требуют тех или иных модификаций. Это может быть вызвано изменением формы номеров служащих, например с 12345 на 12-345, или изменением формата отчетов. В других случаях требуется более значительная модификация, как, например, добавление к программе новых возможностей, выполнение дополнительных вычислений или изменение выполняемых действий при вводе каких-то специальных данных. Структура программы может оказывать существенное влияние на то, легко или трудно реализовать эти изменения в программе. Здесь даны некоторые рекомендации, которые облегчают внесение изменений и программы.

(а) Детально документируйте программы. Программист, вносящий изменения в программу, должен знать не только, где выполняется определенная функция, но также среду, в которой она работает.

Например, если необходимо выйти из цикла DO и вы намереваетесь позднее использовать значение индекса J, то программист, модифицирующий вашу программу, должен быть предупрежден о том, что нельзя модифицировать значение J. Наиболее подходящий путь для выполнения этого условия — расположить инструкции, зависящие от J, непосредственно в конце цикла. Если это невозможно, следует поместить комментарий в конце цикла, предупреждающий, что значение J не должно изменяться.

Мы уже установили, что каждой подпрограмме и каждой достаточно крупной секции программы должен предшествовать список наиболее важных переменных. Этот список необходим, в частности, для того, чтобы подпрограмму было впоследствии легко изменять.

(б) Не используйте хитрых конструкций. Мы предупреждали, что с точки зрения эффективности применение хитрых, или или «заумных», программ на практике дает сомнительную эконо-

мию. С точки зрения изменяемости программы практика применения «заумных» приемов программирования вообще вредна.

(в) Структура программы должна быть модульной. Это означает, что программу следует аккуратно разделить на отдельные модули, а входы и выходы каждого модуля следует четко описать. Тогда пользователь, которому понадобилось модифицировать программу, сможет быстро найти соответствующие секции, в которые необходимо внести изменения, или произвести расширения, согласуя их с существующими участками программы.

(г) Пишите таблично-управляемые программы. Предположим, что вы пишите редактор текста, т. е. программу, которая вводит строки текста и выводит ту же самую информацию в формате «печатной страницы». Программа может полностью заполнять строки (т. е. сливать текст из различных входных строк, печатая столько слов, сколько может поместиться в каждой выводной строке) или не полностью; выравнивать текст справа (вставлять дополнительные пробелы, чтобы все выводные строки были одной и той же длины, подобно строкам этой книги) или не выравнивать; печатать в каждой строке или через строку; пропускать *n* строк и т. п. Для управления этой программой вы можете разрешить пользователям добавлять команды редактирования, которые либо будут выравнивать текст справа, либо нет; будут пропускать *n* строк, определять длину входной или выводной строки и т. д.

Пусть эти команды будут вставляться на строках, содержащих точки с двумя последующими буквами, характеризующими команду.

Ниже приведен список таких строк:

```
.JU   ВЫРАВНЯТЬ ВПРАВО
.NJ   НЕ ВЫРАВНИВАТЬ ВПРАВО
.SS   ПЕЧАТЬ В КАЖДОЙ СТРОКЕ
.DS   ПЕЧАТЬ ЧЕРЕЗ СТРОКУ
.SP N ПРОПУСК N ПУСТЫХ СТРОК
.OL N СОЗДАНИЕ ВЫВОДНЫХ СТРОК ДЛИНОЙ N
.IL  N ПЕРВЫХ ЛИТЕР ИЗ КАЖДОЙ ВХОДНОЙ СТРОКИ
```

Одни действия должны выполняться для всех команд редактирования, другие — только тогда, когда присутствуют какие-то отдельные команды редактирования. При использовании команд, влияющих на формат вывода, вы хотите, например, печатать текущую выводную строку в старом формате, чтобы эта команда оперировала лишь с последующими выводными строками. Проиллюстрируем сказанное на примере такой таблицы:

КОМАНДЫ	ПЕЧАТАТЬ ЛИ ТЕКУЩУЮ ВЫВОДНУЮ СТРОКУ?	ТРЕБУЕТ ЛИ КОМАНДА ЧИСЛА N?	ПОДРАЗУМЕВАЕМОЕ ЗНАЧЕНИЕ N
.JU	ДА	НЕТ	—
.NJ	ДА	НЕТ	—
.SS	ДА	НЕТ	—
.DS	ДА	НЕТ	—
.SP	ДА	ДА	1
.OL	ДА	ДА	80
.IL	НЕТ	ДА	72

Эта таблица может быть естественно представлена серией массивов, один из которых содержит команды, второй 1 и 0 (или .TRUE. и .FALSE.), что означает «ДА» или «НЕТ», и последний массив — значение N (в гл. 3 мы расскажем о представлении литерных данных в Фортране). Назовем эти массивы соответ-

```
C        ПОИСК КОМАНДЫ РЕДАКТИРОВАНИЯ ТЕКСТА
      ...
      ...
C        ПУСТЬ КОМАНДА БЫЛА НАЙДЕНА В COMAND(K)
C
C        ПРОВЕРИМ, ДОЛЖНА ЛИ НАЧАТЬСЯ НОВАЯ СТРОКА
      IF (PRLIN(K) .EQ. 1) CALL PRINT(LINE,LINLEN)
C        ПРОВЕРИМ, ЕСТЬ ЛИ ЗДЕСЬ ЧИСЛО
      IF (NUMBER(K) .EQ. 0) GO TO 2
         CALL GETNUM (INPUT,N,FOUND)
         IF (.NOT.FOUND) N = DEFLT(K)
C
C        ТЕПЕРЬ ОБЩИЕ ДЕЙСТВИЯ ПРОДЕЛАНЫ;
C        ЗАЙМЕМСЯ КАЖДОЙ КОМАНДОЙ ОТДЕЛЬНО
    2 GO TO (10,15,20,25,30,40,50),K
C           JU NJ SS DS SP OL IL
C
C        .JU - НАЧАТЬ ВЫРАВНИВАНИЕ СПРАВА
   10 ADJSW = .TRUE.
      GO TO 100
C
C        .NJ - ПРЕКРАТИТЬ ВЫРАВНИВАНИЕ
   15 ADJSW = .FALSE.
      GO TO 100
C
C        .SP N - ПРОПУСТИТЬ N СТРОК
   30 DO 35 I = 1,N
         CALL PRINT (BLANKS,1)
   35 CONTINUE
      GO TO 100
C
      ...
      ...
C        ОБРАБОТКА ЗАКОНЧЕНА; ВВЕСТИ НОВУЮ КАРТУ
  100 CALL GETCRD
      RETURN
      END
```

Рис. 1.4. Программа, управляемая таблицей.

ственно COMAND, PRLINE, NUMBER и DEFLT. Продемонстрируем на рис. 1.4 фрагмент программы, который управляет выводом в соответствии с добавляемыми командами редактирования.

Заметим, что обработка каждой отдельной команды довольно короткая, так как мы уже выполнили все общие действия. Чтобы добавить новую команду, нужно просто сделать новую строку в таблице (расширяя массивы) и придумать свой код для этой новой команды редактирования. Аккуратное заполнение всех массивов не позволит вам забыть о каких-либо необходимых действиях, связанных с новой командой. Такой стиль программирования называют «таблично-управляемым», так как элементы таблицы управляют работой алгоритма.

Основное преимущество таблично-управляемых программ состоит в том, что такие программы легко модифицировать. Кроме того, благодаря простой реализации зачастую довольно сложного набора решений программу очень легко читать. Вследствие того что структура программы относительно проста, уменьшается вероятность ошибок, и поэтому программу легко отлаживать. В следующем разделе мы увидим, как можно сделать таблично-управляемые программы более наглядными.

1.8. Разделяемые ячейки памяти: инструкция EQUIVALENCE

Инструкция EQUIVALENCE позволяет двум или более переменным разделять одну и ту же ячейку памяти. Конечно, так как одна ячейка памяти в данный момент может содержать только одно значение, инструкция EQUIVALENCE в действительности определяет синонимы, т. е. различные имена, посредством которых осуществляется доступ к одной ячейке памяти. Инструкция EQUIVALENCE имеет следующую форму записи:

EQUIVALENCE (*имена* 1) [, (*имена* 2), и т. д.]

где каждый список *имена* должен содержать перечисление простых переменных и элементов массивов с постоянными индексами.

Пример:

```
EQUIVALENCE (KOUNT, ICOUNT)
KOUNT = 10
ICOUNT = ICOUNT + 1
WRITE (-,-) KOUNT
```

Действие: Декларируется, что KOUNT и INCOUNT ссылаются на одну и ту же ячейку памяти. Значение, печатаемое инструкцией WRITE, будет равно 11;

так как KOUNT и INCOUNT эквивалентны, третья инструкция имеет то же действие, что и

KOUNT=KOUNT + 1

Замечание: Инструкция EQUIVALENCE неисполняемая и должна располагаться в программном модуле после всех специфицирующих инструкций, но до первой исполняемой инструкции.

При использовании инструкции EQUIVALENCE необходимо соблюдать два условия. Во-первых, напомним, что «целые» и «вещественные числа» означают лишь тот или иной способ интерпретации битовых цепочек, и на большинстве ЭВМ двоичное представление целого числа существенно отличается от двоичного представления того же вещественного числа. Например, следующий сегмент программы выглядит аналогичным приведенному выше примеру.

```
REAL COUNT
INTEGER KOUNT
EQUIVALENCE (COUNT, KOUNT)
KOUNT =.10
COUNT = COUNT + 1.0
WRITE (-,-) KOUNT
```

Полученный ответ будет различен для различных компиляторов, но он будет, почти наверняка, *не* равен 11! На IBM 360, например, результат равен 1091567616. В гл. 6 мы рассмотрим внутреннее представление данных, и тогда вы сможете понять, почему получается такой странный ответ. В связи с тем что целые и вещественные представления редко совпадают, следует соблюдать осторожность, делая эквивалентными переменные различного типа.

Второе предостережение, касающееся использования инструкции EQUIVALENCE: следует помнить, что массив представляется рядом *смежных* ячеек памяти. Поэтому эквивалентность между некоторыми элементами двух массивов предполагает также эквивалентность между остальными элементами этих массивов. Чтобы понять, что это значит, рассмотрим массивы A и B размерами 10 и 6 соответственно. Положим, что в программе встретились инструкции

```
REAL A(10), B(6)
EQUIVALENCE (A(3), B(1))
```

Нам известно, что A(4) непосредственно следует в памяти за A(3) и что B(2) следует за B(1). Более того, B(1) и A(3) ссылаются на одну и ту же ячейку памяти; назовем эту ячейку X. Ячейка памяти, непосредственно следующая за X, принадле-

жит A(4), так как A(4) непосредственно следует за A(3). Ячейка, расположенная после X, также принадлежит B(2), так как B(2) непосредственно следует за B(1). Поэтому A(4) и B(2) ссылаются на одну и ту же ячейку, расположенную непосредственно за X. Отсюда эквивалентность между A(3) и B(1) приводит автоматически к эквивалентности между A(4) и B(2), A(5) и B(3) и т. д. Представим эти отношения эквивалентности в наглядной форме:

A(1)	A(2)	A(3)	A(4)	---	A(7)	A(8)	A(9)	A(10)
		B(1)	B(2)	---	B(5)	B(6)		

Можно сделать эквивалентными более двух массивов, но новый массив не должен «растягивать» или «сжимать» другие массивы, т. е. не должно нарушаться подразумеваемое выравнивание элементов массивов, между которыми уже установлена эквивалентность. Например, следующая запись недопустима:

```
REAL A(10), B(6), C(4)
EQUIVALENCE (A(3), B(1)), (C(1)), (C(4), B(1))
```

Приведенная запись нелогична, так как она требует, чтобы элемент B(1) находился одновременно в двух местах: в A(3) и A(4). Инструкция EQUIVALENCE, включающая C, неправильно определяет место B.

В инструкции EQUIVALENCE можно ссылаться на элемент многомерного массива с помощью одиночного индекса (константы). Этот индекс идентифицирует номер элемента относительно первого элемента массива в соответствии с принятым в Фортране методом хранения массивов по столбцам. Предположим, что имеется массив A размером 20 на 25 и вы хотели бы сделать эквивалентными первые 100 элементов этого массива массиву B размером 100, а последние 400 элементов массива A сделать эквивалентными массиву C размером 400. Следующая инструкция дает правильное решение этой задачи, здесь не нужно думать о том, какой элемент массива A является 101-м.

```
EQUIVALENCE (A(1), B(1)), (A(101), C(1))
```

Может показаться, что инструкция эквивалентности имеет сравнительно ограниченное применение, поскольку на первый взгляд нет необходимости ссылаться к одной и той же величине посредством разных имен. Однако существует несколько довольно важных применений инструкции EQUIVALENCE.

Первый пример наиболее простой. Два программиста, работая над большой программой, планируют объединить свои раздельные модули. Они находят, однако, что для обозначения одной

и той же переменной один из них использовал имя QY, а второй— имя QTY. Посредством эквивалентности это различие может быть легко устранено. (Однако это не способствует удобству чтения программы, в которой ссылка к одному и тому же значению происходит по двум совершенно различным именам. Подобный выход из положения называют «заплаткой» в противоположность «решению». Конечно, два программиста, работающие над одной и той же задачей, редко допускают такие промахи.)

Второе применение. Эквивалентность может сэкономить память, позволяя использовать одну и ту же область памяти в двух независимых целях. Приведем пример. Для определения того, к какому семейству животных принадлежит неизвестный вид, ученые используют следующий метод. Во-первых, они устанавливают размеры представителей этого вида. Измерения могут включать длину, вес, размах крыла, количество ног, наличие щупальцев и т. п. Затем они отбрасывают редкие максимальные и минимальные значения каждого измерения как отклонения от нормы и подсчитывают среднее от оставшихся значений. Так вырабатывается портрет (прототип) «типичного», или «среднего», представителя вида. Наконец, ученые сравнивают этот портрет с портретами известных видов и решают, какой из известных видов ближе всего к неизвестному.

В программе, реализующей эту методику, необходимо иметь две большие области данных: первую — для хранения размеров всех представителей нового вида и вторую — для запоминания характерных размеров всех известных видов. Однако эти области используются независимо; когда определение средних характеристик (для неизвестного вида) завершено, область памяти, содержащая результаты отдельных измерений, больше не нужна. Эта область может быть общей с областью, предназначенной для хранения прототипов известных видов. Это взаимно исключающее разделение памяти может быть выполнено инструкцией EQUIVALENCE, пример которой показан ниже:

```
INTEGER SAMPLS (100, 10), PROFIL (100, 50)
EQUIVALENCE (SAMPLS, PROFIL)
```

Третье применение эквивалентности — обеспечить описательные имена для частей массива. Например, при написании программы усреднения очков, набранных студентами, и вычисления отметок необходимо знать отсекающие значения — наименьшее количество очков для отметок A, B, C и D. Для удобства поместим эти значения в массив CUTOFF размером 4[1]. Однако если в

[1] Смысл этого примера трудно понять, не зная английского языка. Допустив, что в Фортране разрешены имена, использующие русский алфавит, переведем эти две инструкции так:

```
IF (НАБР-ОЧКИ. GE. ПРОХ-БАЛЛ) —
IF (НАБР-ОЧКИ. GE. ОТСЕК (4)) — —
```

Прим. ред.

некоторой точке программы необходимо подсчитать количество студентов, получивших проходные отметки, то инструкция вида

IF (SCORE .GE. PASS) — —

дает читателю программы больше информации, чем

IF (SCORE .GE. CUTOFF(4))— —

Инструкция EQUIVALENCE вида

EQUVALENCE (PASS, CUTOFF(4))

разрешит эту проблему.

Следующее применение эквивалентности — если, например, мы имеем дело со значительным количеством простых переменных, которые необходимо занулять несколько раз во время выполнения программы, можно каждую из этих переменных сделать эквивалентной с каким-то элементом массива и очищать весь массив в цикле DO. Следующий пример иллюстрирует, как это можно сделать:

```
      REAL COUNT,SUM,AVG,STDEV,ZEROS(4)
      EQUIVALENCE (ZEROS(1),COUNT), (ZEROS(2),SUM), (ZEROS(3),AVG),
     X   (ZEROS(4),STDEV)
```

Инструкции эквивалентности применяют и в случае использования многочисленных имен для обращения к различным частям массива. В рассмотренной выше таблично-управляемой программе работа выполнялась с четырьмя различными массивами. В некоторых случаях удобно работать с одним массивом из четырех столбцов и иметь возможность обращаться к столбцам по индивидуальным именам. Так как массивы в Фортране хранятся в памяти по столбцам, можно установить эквивалентность между вектором и столбцом массива. Для примера таблично-управляемой программы можно написать

```
      INTEGER CONTAB(7,4), COMND(7),PRLINE(7),NUMBER(7),DEFLT(7)
      EQUIVALENCE (CONTAB(1,1),COMND), (CONTAB(1,2),PRLINE),
     X   (CONTAB(1,3),NUMBER), (CONTAB(1,4),DEFLT)
```

Таким образом, можно использовать CONTAB, как будто мы работаем с каким-то двумерным массивом, чтобы, например, инициализировать данные или изменить значения целой строки. Можно также обращаться к элементам CONTAB с помощью более удобных для чтения имен COMND или DEFLT.

Инструкция EQUIVALENCE очень мощная, возможно, даже слишком мощная для некоторых программистов. Они считают эту инструкцию очень «остроумной», поскольку она позволяет изменить значения переменных даже без обращения к ним по

имени; иногда программисты получают удовольствие, применяя различные хитрости с инструкцией EQUIVALENCE, которые зависят от диалекта. Мы не можем запретить такое применение эквивалентности, но подобные игры с этой инструкцией не свидетельствуют о зрелости программиста.

1.9. Переработанный пример

Теперь настало время пересмотреть пример программы, с которого началась данная глава. Он будет проработан методом пошагового уточнения. Чтобы воспользоваться этим методом, вначале разложим работу программы на небольшое число самых крупных шагов. Затем рассмотрим каждый шаг и уточним его, делая из одного шага несколько меньших шагов. Продолжаем разбивать каждый шаг на несколько меньших шагов, пока весь алгоритм не будет представлен ясными, простыми шагами.

Чтобы вспомнить сущность исходной программы, вам следует просмотреть ее заголовок в разд. 1.2. Повторим четыре основных шага работы программы:

1. Ввести карты, содержащие идентификационные (инвентарные) номера книг и имеющееся количество книг.
2. Отсортировать данные из входных карт в порядке возрастания инвентарных номеров книг.
3. Ввести карты, содержащие номера книг и количество проданных книг.
4. Напечатать суммарный отчет: номера книг имеющихся в наличии и их количества.

Эти пункты составляют первый уровень в нашем методе пошагового уточнения. Теперь для дальнейшей детализации первый шаг разобьем на следующие шаги:

1.1. Определить, сколько перфокарт (имеющихся в наличии книг) должно быть считано.
1.2. Если не будет считано ни одной перфокарты, останов.
1.3. Ввести и обработать перфокарты.

Шаги 1.1 и 1.2 существенно детализированы, чтобы их можно было непосредственно запрограммировать. Шаг 1.3, однако, требует дальнейшего уточнения.

Разобьем его на шаги с 1.3.1 по 1.3.3 для каждой перфокарты имеющейся в наличии книги.

1.3.1. Если для нового номера книги нет свободной памяти, то напечатать сообщение об ошибке и прекратить обработку.
1.3.2. Ввести BOOKNO(NENTRY) и QTY(NENTRY).

1.3.3. Проверить, не повторяется ли этот номер книги. Если да, напечатать сообщение об ошибке и проигнорировать данный номер. Если нет, увеличить NENTRY.

Так как карты с номерами книг не упорядочены, мы не можем применить бинарный поиск для шага 1.3.3. Вместо этого будем использовать обычный последовательный поиск. Для второго шага применим алгоритм быстрой сортировки, чтобы отсортировать данные. Мы не детализируем шаг 2, так как его уточнение в точности соответствовало бы алгоритму, приведенному уже в разд. 1.6.

Шаг 3 включает несколько уточнений. Мы представим все уточнения сразу, поскольку многие из них соответствуют приведенным на шаге 1. Можно описать пошаговую детализацию в указанном порядке, если вы себе ясно представляете логическое развитие шага 3. Важная особенность пошагового уточнения состоит в том, что это помогает вам составлять программу. Не стоит обращать внимание на некоторые технические ограничения, присущие процессу пошагового уточнения; постарайтесь извлечь из него как можно больше.

3.1. Определим, сколько считано карт с номерами проданных книг.
3.2. Если не считано ни одной перфокарты, останов.
3.3. Обработаем карты проданных книг.
  DO для каждой считанной карты проданных книг:
    Ввод CBOOK и CQTY, данные для одной книги.
    Поиск CBOOK в массиве BOOKNO.
      IF NENTRY=0, поиск окончен; номер не найден.
      DO для J=1 до NENTRY.
        IF BOOKNO(J)=CBOOK, поиск окончен; номер найден.
        IF BOOKNO(J) > CBOOK, поиск окончен; номер не найден.
      Если цикл оканчивается нормально, поиск окончен; номер не найден.
    Если номер не найден, печать сообщения об ошибке и ввод следующей перфокарты.
    Если найден, то обработать данные о продаже книги.
      Если проданное количество меньше или равно имеющемуся количеству, уменьшить имеющееся количество соответствующим образом; иначе установить имеющееся количество в нуль и напечатать сообщение.

Шаг 4 включает только печать содержимого массивов BOOKNO и QTY.

Теперь можно избавиться от избыточных инструкций программы с целью упрощения алгоритма. Выполняются два поиска: на шаге 1 и на шаге 3. По назначению они подобны, но не идентичны. Их можно свести в общую подпрограмму для выполнения

```
      SUBROUTINE LOOKY(TABLE,TABSIZ,ENTRY,SORTED,FOUND,WHERE,DBG)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C  ПОДПРОГРАММА ПОИСКА В ТАБЛИЦЕ
C
C  ВХОДНЫЕ ЗНАЧЕНИЯ:
C    «TABLE»-МАССИВ РАЗМЕРОМ «TABSIZ», В КОТОРОМ ИДЕТ ПОИСК
C    ЭЛЕМЕНТА «ENTRY». ЕСЛИ ТАБЛИЦА УПОРЯДОЧЕНА ПО
C    ВОЗРАСТАНИЮ, ТО ПЕРЕМЕННАЯ «SORTED» ИМЕЕТ ЗНАЧЕНИЕ TRUE;
C    В ПРОТИВНОМ СЛУЧАЕ FALSE.
C    «DBG» УПРАВЛЯЕТ ОТЛАДОЧНЫМ ВЫВОДОМ; НЕНУЛЕВОЕ
C    ЗНАЧЕНИЕ DBG ОЗНАЧАЕТ ТРАССИРОВКУ ПРИ ВЫХОДЕ ИЗ
C    ПОДПРОГРАММЫ.
C
C  УСЛОВИЯ ВЫХОДА:
C    ЕСЛИ ЭЛЕМЕНТ НАЙДЕН, «FOUND» ИМЕЕТ ЗНАЧЕНИЕ TRUE,
C    «WHERE»-ЭТО ИНДЕКС ЭЛЕМЕНТА ENTRY В МАССИВЕ TABLE.
C    ЕСЛИ ЭЛЕМЕНТ НЕ БЫЛ НАЙДЕН, ТО «FOUND» ИМЕЕТ ЗНАЧЕНИЕ
C    FALSE, А ЗНАЧЕНИЕ «WHERE» НЕ ИЗМЕНЯЕТСЯ.
C  ПРОГРАММА ВЫПОЛНЯЕТ ПРОСТОЙ ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК.
C  ЕСЛИ TABLE ОТСОРТИРОВАНА, ТО ПОИСК ПРОДОЛЖАЕТСЯ ТОЛЬКО
C  ДО ТЕХ ПОР, ПОКА НЕ БУДЕТ НАЙДЕН ЭЛЕМЕНТ, БОЛЬШИЙ, ЧЕМ
C  ENTRY; ЕСЛИ НЕ ОТСОРТИРОВАНА, НЕОБХОДИМ ПРОСМОТР ВСЕЙ
C  ТАБЛИЦЫ.
C
C    ПРОГРАММИСТ:---              ДАТА:---
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      INTEGER TABSIZ, TABLE(TABSIZ), ENTRY, WHERE, DBG, DBOUT
      LOGICAL SORTED, FOUND
      DATA DBOUT/6/
C         ПРЕДПОЛОЖИМ, ЧТО ЭЛЕМЕНТ НЕ БУДЕТ НАЙДЕН
      FOUND = .FALSE.
C
C         ЕСЛИ ТАБЛИЦА ПУСТА, НЕ НУЖНО ИСКАТЬ
      IF (TABSIZ .LE. 0) RETURN
C
C         ЦИКЛ ДЛЯ АНАЛИЗА КАЖДОГО ЭЛЕМЕНТА
      DO 10 I = 1,TABSIZ
         IF (TABLE(I) .EQ. ENTRY) GO TO 50
         IF (TABLE(I) .LT. ENTRY) GO TO 10
         IF (SORTED) GO TO 99
   10 CONTINUE
C
C         ЕСЛИ ЦИКЛ ОКАНЧ. НОРМАЛЬНО, ТО ЭЛЕМЕНТ НЕ НАЙДЕН
      GO TO 99
C
C         ПРИХОДИМ СЮДА, ЕСЛИ ЭЛЕМЕНТ НАЙДЕН
   50 FOUND = .TRUE.
      WHERE = I
C
C         ОБЩАЯ ТОЧКА ВОЗВРАТА; УСТАНАВЛИВАЕТСЯ FOUND
   99 IF (DBG .EQ. 0) RETURN
      IF (FOUND) WRITE (DBOUT,101) ENTRY, WHERE
      IF (.NOT. FOUND) WRITE (DBOUT,102) ENTRY, TABSIZ
      RETURN
  101 FORMAT (' EXIT FROM SUBR. LOOKY: ITEM', I8, ' FOUND AT POS.', I8)
  102 FORMAT (' EXIT FROM SUBR. LOOKY: ITEM', I8, ' NOT FOUND IN TABLE',
     X  ' TABLE SIZE IS', I6)
      END
```

Рис. 1.5. Подпрограмма поиска в таблице. (101: «выход из подпрогр. LOOKY: элемент», «найден в позиции»; 102: «выход из подпрогр. LOOKY: «элемент», «не найден в таблице».)

табличного поиска. Введем параметры (NENTRY — 1) или NENTRY и BOOKNO (NENTRY) или CBOOK. Можно также добавить еще один параметр, чтобы показать, отсортирован массив или нет, так как только для поиска на шаге 3 требуются упорядоченные данные. Подпрограмма приведена на рис. 1.5.

Процесс сортировки, хотя он встречается только однажды, может быть представлен в виде подпрограммы, чтобы обеспечить модульность рассматриваемой программы. Кодирование этой подпрограммы выполняется в соответствии с основными принципами разд. 1.6.2, и поэтому мы не приводим ее листинг.

В программе имеется также несколько точек, из которых выдаются сообщения об ошибках. Эти инструкции следует выделить из программы, поскольку подобные сообщения не имеют прямого отношения к программе, как таковой, и могут затруднить ее чтение. Кроме того, этот модуль для первоначальной программной отладки можно запрограммировать попроще (только печать номеров сообщений об ошибке), а позднее его можно «приукрасить» вразумительными текстами сообщений об ошибках. Подпрограмма в качестве параметров будет принимать номер ошибки и указание о том, фатальна ли ошибка (когда обработка не может быть продолжена), или это просто предупреждение (обработка может быть продолжена). Третьим параметром служит связанный с ошибкой инвентарный номер книги или 0 для тех ошибок, которые не имеют отношения к номеру книги. Приведем эту подпрограмму.

```
      SUBROUTINE ERRMSG (ERRNO, FATAL, BOOKNO)
C      ПОДПРОГРАММА, ВЫДАЮЩАЯ СООБЩЕНИЯ ОБ ОШИБКАХ
      INTEGER ERRNO, BOOKNO, OUT
      LOGICAL FATAL
      DATA OUT/6/
C      ВЫВОД СООБЩЕНИЯ ОБ ОШИБКАХ
      WRITE (OUT,1) ERRNO
    1 FORMAT (' ERROR NUMBER ', I3, ' OCCURRED.')
      IF (BOOKNO .NE. 0) WRITE (OUT,2) BOOKNO
    2 FORMAT (' INVOLVING BOOK NUMBER', I10)
C      НАСКОЛЬКО СЕРЬЕЗНА ОШИБКА?
      IF (.NOT. FATAL) RETURN
      WRITE (OUT, 3)
    3 FORMAT (' EXECUTION TERMINATING')
      STOP
      END
```

```
1: 'ОШИБКА НОМЕР', 'ПРОИЗОШЛА'
2: 'СВЯЗАННАЯ С НОМЕРОМ КНИГИ'
3: 'ВЫПОЛНЕНИЕ ПРЕКРАЩАЕТСЯ'
```

Закончив все необходимые подпрограммы, рассмотрим теперь главную программу. В этой программе мы применили одно из наших собственных предложений об использовании сигнальной

перфокарты для отметки конца каждой из двух частей входной колоды. Номер книги —1 будет отмечать конец каждой части. Заметим, что при обращении к ERRMSG мы использовали мнемонические имена переменных, такие, как OVFLOW (ПЕРЕПЛ), DUPL (ДУБЛ), NEWBK (НОВКН) и др. Благодаря этим именам нет необходимости помнить цифровые коды, связанные с различными сообщениями об ошибках. Аналогично мы использовали мнемонические обозначения для указания фатальной или нефатальной ошибки. Полностью программа приведена на рис. 1.6.

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C  ПРОГРАММА ИНВЕНТАРИЗАЦИИ КНИГ
C
C  ЭТА ПРОГРАММА ВВОДИТ НАБОР НОМЕРОВ КНИГ И ИМЕЮЩЕЕСЯ
C    КОЛИЧЕСТВО ЭТИХ КНИГ И ОБРАБАТЫВАЕТ ДАННЫЕ О ПРОДАЖЕ
C    КНИГ В СООТВЕТСТВИИ С ПЕРВОНАЧАЛЬНОЙ ИНВЕНТАРИЗАЦИЕЙ.
C    ПРОГРАММА ПЕЧАТАЕТ ЗАКЛЮЧИТЕЛЬНЫЙ ОТЧЕТ ОБ
C    ИМЕЮЩИХСЯ В НАЛИЧИИ КНИГАХ.
C  ЕСЛИ СДЕЛАНА ПОПЫТКА ПРОДАТЬ БОЛЬШЕ КНИГ, ЧЕМ ИХ ИМЕЕТСЯ
       В НАЛИЧИИ, ПРОГРАММА ПЕЧАТАЕТ СООБЩЕНИЕ.
C
C  ВХОДНЫЕ ДАННЫЕ:
C  СЕРИЯ КАРТ С НОМЕРАМИ ПЕРВОНАЧАЛЬНО ИМЕЮЩИХСЯ
C  КНИГ/КОЛИЧЕСТВА.
C  КОЛОДА КАРТ ОКАНЧИВАЕТСЯ КАРТОЙ С ЧИСЛОМ —1.
C  СЕРИЯ КАРТ С НОМЕРАМИ ПРОДАННЫХ КНИГ/КОЛИЧЕСТВА.
C  КОЛОДА КАРТ ЭТОЙ ГРУППЫ ЗАВЕРШАЕТСЯ КАРТОЙ С ЧИСЛОМ —1.
C
C  ПРОГРАММА СОСТОИТ ИЗ ЧЕТЫРЕХ ФАЗ:
C  1. ВВОД ПЕРВОНАЧАЛЬНЫХ ИНВЕНТАРИЗАЦИОННЫХ КАРТ;
C  2. СОРТИРОВКА НАЧАЛЬНЫХ НОМЕРОВ КНИГ И МАССИВОВ
C     КОЛИЧЕСТВА КНИГ;
C  3. ВВОД КАРТ ДЛЯ ПРОДАННЫХ КНИГ;
C  4. ПЕЧАТЬ ЗАКЛЮЧИТЕЛЬНОГО ОТЧЕТА О КНИГАХ, ИМЕЮЩИХСЯ
C     В НАЛИЧИИ.
C
C  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:
C    LOOKY: ВЫПОЛНИТЬ ПОИСК В ТАБЛИЦЕ;
C    QSORT: ОТСОРТИРОВАТЬ МАССИВ, СОДЕРЖАЩИЙ ДВА СТОЛБЦА;
C    ERRMSG: НАПЕЧАТАТЬ СООБЩЕНИЕ ОБ ОШИБКЕ.
C
C  ОСНОВНЫЕ ПЕРЕМЕННЫЕ:
C  BOOKNO — НОМЕРА КНИГ В СИСТЕМЕ ИНВЕНТАРИЗАЦИИ;
C  QTY    — КОЛИЧЕСТВО ЭТИХ КНИГ;
C           QTY(I)-КОЛИЧЕСТВО BOOKNO(I);
C  BOOKS  — МАТРИЦА, СОСТОЯЩАЯ ИЗ СТОЛБЦОВ BOOKNO И QTY;
C  NENTRY — ТЕКУЩИЙ НОМЕР КНИГИ;
C  CBOOK  — ТЕКУЩАЯ КНИГА, ИСПОЛЬЗУЕМАЯ НА ШАГЕ 3;
C  CQTY   — ТЕКУЩЕЕ КОЛИЧЕСТВО, ИСПОЛЬЗУЕМОЕ НА ШАГЕ 3;
C  MAXBKS — МАКСИМАЛЬНОЕ КОЛИЧЕСТВО НОМЕРОВ КНИГ (200);
C  DBG    — УПРАВЛЕНИЕ ОТЛАДКОЙ: =0 ОТЛАДКА НЕ
C           ПРОИЗВОДИТСЯ, =1 ВЫЗОВЫ ПОДПРОГРАММ, =2 ВЫЗОВЫ
C           ПОДПРОГРАММ И ПЕЧАТЬ ВХОДНЫХ ДАННЫХ, =3 ВЫЗОВЫ
C           ПОДПРОГРАММ, ПЕЧАТЬ ВХОДНЫХ ДАННЫХ, ПЕЧАТЬ
C           ТАБЛИЦЫ ПОСЛЕ СОРТИРОВКИ.
C
C     ПРОГРАММИСТ: ---          ДАТА: ---
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 1.6. Пересмотренная программа инвентаризации книг. (**111**: «элемент №», «книга №», «количество»; **201**: «отсортированная таблица книг»; **301**: «Трассир: книга №, количество»; **351**: «недостающее количество»; **401**: «заключительный отчет», «книга №=имеющееся количество».)

```
      INTEGER BOOKS(200,2),BOOKNO(200),QTY(200),CBOOK,CQTY,MAXBKS,
     X   IN,OUT,WHERE,NENTRY
      EQUIVALENCE (BOOKS(1,1),BOOKNO), (BOOKS(1,2),QTY)
C         НОМЕРА СООБЩЕНИЙ ОБ ОШИБКЕ
      INTEGER OVFLOW,DUPL,NEWBK,OVSOLD,EOF1,EOF2, DBG, DBOUT
      DATA OVFLOW/1/,DUPL/2/,NEWBK/3/,OVSOLD/4/,EOF1/5/,EOF2/6/
C         МНЕМОНИЧЕСКИЕ ИМЕНА ДЛЯ ВЫРАЖЕНИЙ: ОШИБКИ
C         ФАТАЛЬНЫЕ/НЕФАТАЛЬНЫЕ, МАССИВ СОРТИРОВАННЫЙ/НЕСОРТИРОВАННЫЙ
      LOGICAL FOUND, FATAL,SORTED
      DATA FATAL/.TRUE./,SORTED/.TRUE./
      DATA IN/5/,OUT/6/,MAXBKS/200/,DBG/0/,DBOUT/6/
C
C
C     ****ФАЗА 1: ВВЕСТИ НАЧАЛЬНЫЕ ИНВЕНТАРНЫЕ ЗНАЧЕНИЯ****
      NENTRY = 1
C         НЕ ПРЕВЫШЕНЫ ЛИ РАЗМЕРЫ МАССИВОВ?
  100 IF (NENTRY .GT. MAXBKS) CALL ERRMSG (OVFLOW,FATAL,0)
C         ВВОД ПАРЫ НОМЕР КНИГИ/КОЛИЧЕСТВО
      READ (IN,110,END=190) BOOKNO(NENTRY), QTY(NENTRY)
  110 FORMAT (2I10)
      IF (DBG .GT. 1) WRITE (DBOUT,111) NENTRY, BOOKNO(NENTRY),
     X                                  QTY(NENTRY)
  111 FORMAT (' ENTRY NO.', I8, ' BOOK NO.', I10, ' QUANTITY', I10)
C         КОНЕЦ ДАННЫХ?
      IF (BOOKNO(NENTRY) .EQ. -1) GO TO 200
C
C         ДУБЛИРУЮТСЯ ЛИ НОМЕРА КНИГ?
  120 CALL LOOKY (BOOKNO, NENTRY-1, BOOKNO(NENTRY), .NOT. SORTED,
     X        FOUND, WHERE, DBG)
      IF (.NOT. FOUND) NENTRY = NENTRY + 1
      IF (      FOUND) CALL ERRMSG (DUPL, .NOT.FATAL, BOOKNO(NENTRY))
      GO TO 100
C


C         ПРИХОДИМ СЮДА, ВСТРЕТИВ КОНЕЦ ФАЙЛА
  190 CALL ERRMSG (EOF1,FATAL,0)
C
C     **** ФАЗА 2: СОРТИРОВКА ДАННЫХ.****
C             КОНЕЦ ДАННЫХ; СБРОСИТЬ „-1“
      NENTRY = NENTRY-1
  200 CALL QSORT (BOOKS, MAXBKS, NENTRY)
      IF (DBG .GT. 2) WRITE (DBOUT,201) (BOOKS(I), I=1,NENTRY)
  201 FORMAT(' SORTED BOOK TABLE:'/ 10(1X,I10))
C
C     **** ФАЗА 3: ОБРАБОТКА ЗАПРОСОВ НА ПРОДАЖУ ****
C
  300 READ (IN,110,END=390) CBOOK,CQTY
      IF (DBG .GT. 1) WRITE (DBOUT,301) CBOOK, CQTY
  301 FORMAT (' TRANSACTION TRACE: BOOK NO., QUANTITY:', 2I10)
C         КОНЕЦ ДАННЫХ?
      IF (CBOOK .EQ. -1) GO TO 400
C
C         НАЙТИ НОМЕР КНИГИ, ЧТОБЫ ОБРАБОТАТЬ ЗАПРОС
      CALL LOOKY (BOOKNO, NENTRY, CBOOK, SORTED, FOUND, WHERE, DBG)
      IF (FOUND) GO TO 350
C
C         НОМЕР КНИГИ ДОЛЖЕН БЫЛ БЫТЬ ВВЕДЕН НА ПЕРВОЙ ФАЗЕ
C         ОШИБКА, ЕСЛИ КНИГИ НЕТ. ИГНОРИРУЕТСЯ ЗАПРОС НА ПРОДАЖУ
          CALL ERRMSG (NEWBK,.NOT.FATAL,CBOOK)
          GO TO 300
C
C         КНИГА НАЙДЕНА; СМОТРИМ, ДОСТАТОЧЕН ЛИ ЗАПАС
  350 IF (QTY(WHERE) .GE. CQTY) GO TO 360
C         ЗАПРОШЕНО БОЛЬШЕ, ЧЕМ В НАЛИЧИИ; ВЫВЕСТИ
C         СООБЩЕНИЕ И УКАЗАТЬ, СКОЛЬКО НЕ ХВАТАЕТ
          CALL ERRMSG (OVSOLD,.NOT.FATAL,CBOOK)
          NQTY = CQTY - QTY(WHERE)
          WRITE (OUT, 351) NQTY
  351     FORMAT(' QUANTITY NOT SHIPPED IS ', I10)
          QTY(WHERE) = 0
          GO TO 300
C
```

Продолжение Рис. 1.6.

```
C     ПРАВИЛЬНЫЙ ЗАПРОС
  360 QTY(WHERE) = QTY(WHERE) - CQTY
      GO TO 30)
C     НЕПРЕДВИДЕННЫЙ КОНЕЦ ВВОДА В ФАЗЕ 3;
C     СЧИТАТЬ, ЧТО ВСТРЕТИЛАСЬ „-1" КАК ПРИЗНАК КОНЦА ЭТИХ ДАННЫХ
  390 CALL ERRMSG (EOF2,.NOT.FATAL,0)
C
C       **** ФАЗА 4: ВЫВОД ОТЧЕТА ****
C
  400 IF (NENTRY .LE. 0) GO TO 500
      WRITE (OUT,4)1) (BOOKNO(I),QTY(I), I=1,NENTRY)
  401 FORMAT ('1ENDING INVFNTORY REPORT'///
     X     5X, 'BOOK NUMBER = QUANTITY ON HAND'/
     X     5(3X, I10, '=', I10, ';') )
C
  500 STOP
      END
```

Продолжение Рис. 1.6.

Ссылки на литературу

Сообщение Дейкстры «О пагубности инструкции GO TO» появилось в *Communications ACM*, **11**, 3, стр. 147—148. Алгоритм быстрой сортировки впервые был описан Хоором в статье «Быстрая сортировка», *Computer Journal*, **5**, I, стр. 10—15. Анализ времени выполнения этого алгоритма, выполненный Кнутом, можно найти в книге Д. Кнута: Искусство программирования для ЭВМ. Том 3: Сортировка и Поиск.— М.: Мир, 1978.

Глава 2

ПРОЦЕДУРЫ

Умение широко применять подпрограммы — один из первых этапов в развитии профессиональных навыков программирования. В гл. 0 и 1 было показано, что применение процедур повышает эффективность программы, делает ее более удобной для чтения. Такую программу проще писать и отлаживать, с ней легче работать. В этой главе расширяются основы, которые были вами уже усвоены, и объясняются более сложные свойства фортранных подпрограмм.

2.1. Элементарные характеристики процедур Фортрана

В языке Фортран допускаются процедуры двух типов функции и подпрограммы. Функция — это процедура, которая всегда возвращает один результат, зависящий от значений передаваемых ей параметров (аргументов). Подпрограмма применяется в более общих случаях и может возвращать несколько результатов в вызывающую программу или даже ничего не возвращать.

Процедура Фортрана состоит из последовательности инструкций, первая из которых указывает ее имя, далее следует набор деклараций, тело процедуры и в конце инструкция END. В первой инструкции процедуры указывается также ее тип (функция или подпрограмма) и набор формальных параметров. В декларациях содержится информация о типе размерности каждого параметра и каждой локальной переменной.

Номера (метки) инструкций, входящих в состав программного модуля (главной программы или процедуры), рассматриваются как локальные в этом модуле. Поэтому в разных процедурах можно использовать идентичные метки. Инструкции, входящие в данный программный модуль, могут содержать ссылки на метки только тех инструкций, которые находятся внутри этого же модуля.

Переменные и массивы, не являющиеся формальными параметрами, по умолчанию считаются локальными по отношению к модулю, в котором они определены. Это означает, что переменной K, встретившейся в одном программном модуле, отво-

дится ячейка памяти, отличная от той, которая будет выделена переменной K, встретившейся в каком-либо другом программном модуле, если, конечно, специально не оговорено противное. В разд. 2.6 мы покажем, как переменные и массивы могут быть объявлены глобальными. В приведенных ниже примерах все переменные и массивы будут формальными параметрами или будут локальными по отношению к модулю, в котором они объявлены.

2.1.1. Функции

Функция определяется как последовательность инструкций, первая из которых имеет следующий вид:

[*тип*] FUNCTION *имя (список-форм-парам)*

Заканчивается функция инструкцией END. Тип функции может быть INTEGER, REAL, LOGICAL, или он может быть опущен. Если тип функции явно не указан, то он определяется по первой букве ее *имени* (имена, начинающиеся с букв I, J, K, L, M, N, указывают на функцию целого типа; в противном случае тип функции считается вещественным). Список формальных параметров состоит из набора переменных и массивов, связанных с соответствующими фактическими параметрами переменными и массивами.

Вызов функции происходит при выполнении арифметического или логического выражения, содержащего имя функции, за которым в скобках следует соответствующий список фактических параметров. При вычислении функции ее имени присваивается значение результата; имя функции используется как переменная. Последнее значение, присвоенное имени функции, рассматривается затем как результат вычисления функции. Возврат в вызывающую программу происходит, когда выполняется инструкция RETURN внутри функции.

Функции следует применять лишь в том случае, когда необходимо передать в вызывающую программу только одно число — результат вычисления функции. Хотя по правилам Фортрана явно не запрещено возвращать дополнительные результаты путем изменения значений параметров, этого делать не рекомендуется. В качестве примера рассмотрим следующую тривиальную функцию F. Она возвращает значение I+1 для заданного значения параметра I.

```
INTEGER FUNCTION F(I)
INTEGER I
I=I+1
F=I
RETURN
END
```

При вычислении функции такого вида изменяется и значение ее параметра. Посмотрим на результат, когда функция F вызывается в инструкции:

```
K=F(J)+F(J)+J
```

где J равно 3. При первом обращении к F возвращаемый результат равен 4, и значение J также стало равным 4. При втором обращении к подпрограмме-функции возвращаемый результат равен 5, и J изменилось на 5. Как следствие, K принимает значение 14=(4+5+5), и в качестве побочного эффекта J присваивается значение 5.

Это довольно неприятное обстоятельство, но еще хуже то, что в действительности все может произойти иначе. Некоторые компиляторы распознают и устраняют неэффективности. В таком компиляторе резонно предположить, что запись

```
K=F(J)+F(J)+J
```

эквивалентна

```
K=2*F(J)+J
```

Но в этом случае K получит значение 12=(2*4+4), а не 14 и значением J станет 4, а не 5.

Нижеприведенная рекомендация не является правилом Фортрана. Скорее это правило «хорошего тона» при программировании на Фортране. Однако мы настоятельно советуем его соблюдать:

Функция никогда не должна изменять значения своих параметров.

Следует также избегать выполнения операций ввода/вывода внутри функции. Это ограничение является следствием того, что не всегда можно точно предугадать, произойдет ли обращение к функции. Например, для инструкции вида

```
IF((J.EQ.O) .OR. (F(J).EQ.K)) GO TO 7
```

если J=0, нет необходимости вызывать функцию F, чтобы определить, нужно ли передавать управление инструкции с меткой 7. В некоторых диалектах Фортрана функция F вызывается всегда, в других не вызывается. Если в функции F выполняется чтение перфокарт или вывод на печать, в различных версиях Фортрана могут получиться разные результаты. Во избежание этого будем руководствоваться следующим правилом:

Функция не должна содержать инструкций ввода/вывода (за исключением, конечно, вывода, необходимого при отладке).

2.1.2. Подпрограммы

Подпрограмма определяется как последовательность инструкций, первая из которых

SUBROUTINE *имя* [*(список-форм-парам)*]

и последняя инструкция END. Подпрограмма вызывается по имени при выполнении инструкции

CALL *имя* [*(список-факт-парам)*]

Результаты могут быть возвращены в вызывающую программу путем изменения значений одного или нескольких параметров подпрограммы. Возврат в вызывающую программу происходит, когда внутри подпрограммы выполняется инструкция RETURN.

2.2. Передача параметров

2.2.1. Основные правила связи формальных и фактических параметров

При обращении к функциям или подпрограммам устанавливается связь между каждым фактическим параметром и соответствующим ему формальным параметром. Первый фактический параметр связан с первым формальным параметром, второй — со вторым и т. д. Число фактических параметров должно соответствовать числу формальных параметров.

Фактический параметр может быть переменной, элементом массива, арифметическим или логическим выражением, массивом. (Другие возможности объясняются в последующих разделах.) Формальный параметр может быть переменной или массивом. Если формальный параметр — переменная, соответствующий ему фактический параметр может быть переменной, элементом массива или выражением. Формальному параметру-массиву должен соответствовать массив или элемент массива. Тип (REAL, INTEGER или LOGICAL) каждого фактического параметра должен быть таким же, как и соответствующий ему тип формального параметра. Старайтесь согласовывать размерность формального массива и его размеры с соответствующими этому массиву фактическими параметрами. Однако правила Фортрана требуют только, чтобы размер (общее число компонентов) формального массива не превышал соответствующего ему размера массива — фактического параметра. В том случае, когда в качестве фактического параметра передается элемент массива, считается, что размер массива в подпрограмме (формального массива) совпадает с размером массива, которому этот элемент

принадлежит, уменьшенным на число элементов, предшествующих данному элементу массива. (См. в разд. 0.2, как массивы Фортрана хранятся в памяти.)

2.2.2. Методика обеспечения связи параметров

В соответствии с определением Фортрана, которое было дано Американским национальным институтом стандартов ANSI, фактический параметр, не являющийся выражением, ассоциируется с формальным параметром (или передается в подпрограмму) *по имени*; если фактический параметр — выражение, то связь организуется *по значению*. Смысл терминов передача параметров «по имени» и «по значению» был умышленно оставлен неопределенным, чтобы позволить разработчикам компиляторов выбирать методы реализации, наиболее подходящие для архитектуры конкретной машины. Разработчики Фортрана, предвидя несовпадение результатов, которое может получиться при различных способах передачи параметров, наложили следующие ограничения:

Если два или более формальных параметра ассоциируются с одной и той же переменной, элементом массива, или массивом, то ни одному из этих формальных параметров не может быть присвоено значение.

Формальному параметру, который ассоциируется с выражением, не может быть присвоено значение.

Теперь мы опишем общеупотребимые методы передачи параметров «по имени» и «по значению» и покажем, как могут быть получены различные результаты, если не учитывать сформулированных выше ограничений.

2.2.3. Передача параметров по имени

Ассоциация параметров по имени реализуется обычно одним из двух методов: *по значению/результату* или *по адресу (по ссылке)*. Если связь параметров организована по адресу, то при каждой ссылке на формальный параметр происходит обращение к ячейке памяти, отведенной соответствующему фактическому параметру. Таким образом, каждое присваивание значения формальному параметру есть, по существу, присваивание фактическому параметру. В противоположность этому, если параметры ассоциируются по значению/результату, то значение фактического параметра при входе в подпрограмму заносится в область памяти, отведенную формальному параметру. Присвоить значение формальному параметру означает — занести

это значение в область памяти, предназначенную для формального, а не фактического параметра. Однако при выполнении инструкции RETURN текущее значение формального параметра копируется обратно в область памяти фактических параметров.

Передача параметров по адресу используется во всех диалектах Фортрана, с которыми мы знакомы, в тех случаях, когда формальный параметр является массивом. Почему — это станет понятно, на примере подпрограммы PRLIST:

```
      SUBROUTINE PRLIST (LIST,N)
C** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **
C*    ПЕЧАТЬ ПЕРВЫХ N ЭЛЕМЕНТОВ ОДНОМЕРНОГО МАССИВА LIST,  *
C*    5 ЧИСЕЛ В СТРОКЕ                                      *
C** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **
      INTEGER LIST(500), N, I, PRT
      DATA PRT/6/
      WRITE (PRT,200) (LIST (I), I = 1, N)
200   FORMAT ('1', (20X,5I5) )
      RETURN
      END
```

Если бы мы воспользовались передачей по значению/результату, то при вызове PRLIST 500 элементов фактического параметра-массива необходимо было бы скопировать в область памяти, предназначенную для массива LIST. Эти 500 значений надо было бы затем перенести обратно при выполнении инструкции RETURN. Ясно, что это расточительство как машинного времени, так и оперативной памяти.

Различие результатов, получаемых при передаче параметров по адресу и по значению/результату, демонстрирует программа, приведенная на рис. 2.1.

При передаче параметров по адресу выполнение инструкции присваивания J=J+1 приводит к тому, что фактический параметр M получает значение 2; соответственно формальный параметр K также принимает значение 2. В результате выполнения инструкции WRITE в подпрограмме SUB будет напечатано

J=2 K=2

Посмотрим теперь, что произойдет, если параметры передаются по значению/результату. В этом случае и для J, и для K отводятся ячейки памяти, а при выполнении инструкции CALL J и K будет присвоено значение 1. После выполнения инструкции J=J+1 переменная J получит значение 2, но это не вызывает соответствующего изменения значения K. Соответственно в результате выполнения инструкции WRITE в подпрограмме SUB будет напечатано

J=2 K=1

Различные результаты могут быть также получены при использовании двух различных диалектов Фортрана, в которых связь параметров организована по значению/результату. Обратимся вновь к нашему примеру. Какое значение получит M после

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ПРОГРАММА ДЕМОНСТРИРУЕТ РАЗЛИЧИЯ МЕЖДУ ДВУМЯ МЕТОДАМИ              *
C* ПЕРЕДАЧИ ПАРАМЕТРОВ.                                                *
C* ПРИ ВЫЗОВЕ ПОДПРОГРАММЫ SUB ПЕРЕМЕННАЯ M                            *
C* ИГРАЕТ РОЛЬ ОБОИХ ПАРАМЕТРОВ                                        *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER M, PRT
      DATA PRT/6/
C
C         M ПРИСВАИВАЕТСЯ ЗНАЧЕНИЕ 1, ЗАТЕМ — ВЫЗОВ SUB,
C         В РОЛИ ОБОИХ ПАРАМЕТРОВ M
      M = 1
      CALL SUB (M, M)
C
C         ПРОСЛЕДИТЕ ЗА ЗНАЧ. M. ДАЖЕ В РАЗНЫХ СИСТЕМАХ,
C         ИСПОЛЬЗ. МЕТОД ПО ЗНАЧ., РЕЗУЛЬТАТЫ МОГУТ РАЗЛИЧАТЬСЯ
      WRITE (PRT,200) M
200   FORMAT (10X, 'M=', I1)
      STOP
      END

      SUBROUTINE SUB (J, K)
      INTEGER J, K, PRT
      DATA PRT/6/
C
C         МЕНЯЕМ ЗНАЧ. 1-го ФОРМ. ПАРАМЕТРА. ЕСЛИ АССОЦ.
C         ПО ССЫЛКЕ, ТО МЕНЯЕТСЯ ЗНАЧ. И 2-го ФОРМ. ПАРАМЕТРА
      J = J+1
C
C         СМ. КАК ЭТО ВЛИЯЕТ НА K
      WRITE (PRT,200) J, K
200   FORMAT (11X, 'J=', I1, 3X, 'K=', I1)
      RETURN
      END
```

Рис. 2.1. Сравнение двух методов передачи параметров: по ссылке и по значению.

возврата из подпрограммы? Если вначале в M заносится значение J, а затем значение K, то последнее значение M равно значению K, т. е. 1. Однако если K копируется раньше J, то M принимает значение 2.

2.2.4. Передача параметров по значению

Передача параметров по значению выполняется в диалектах Фортрана более согласованно, чем по имени. Метод передачи по значению, используемый для параметров-выражений, требует, чтобы память для значения параметра была выделена внутри вызывающего (а не вызываемого) программного модуля. Выражение вычисляется до входа в вызываемый модуль, и его значение заносится в специально отведенную область памяти.

Проблемы возникают в том случае, когда фактический параметр — константа. В некоторых диалектах Фортрана в данной ситуации формальный параметр непосредственно ассоциируется с ячейкой памяти, в которой хранится константа. Это может привести к ужасному побочному эффекту. В качестве примера

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ПРОГРАММА ПОКАЗЫВАЕТ, КАК МОЖЕТ ИЗМЕНИТЬСЯ ЗНАЧЕНИЕ КОНСТАНТЫ,      *
C* ПЕРЕДАННОЙ ПОДПРОГРАММЕ В КАЧЕСТВЕ ПАРАМЕТРА,                         *
C* КОТОРОМУ ПРИСВАИВАЕТСЯ ЗНАЧЕНИЕ В ПОДПРОГРАММЕ.                      *
C* УДИВИТЕЛЬНО, НО ФАКТ: ВО МНОГИХ ДИАЛЕКТАХ (НО НЕ WATFIV)             *
C* ПРОГРАММА В РЕЗУЛЬТАТЕ НАПЕЧАТАЕТ                                     *
C* J=2                                                                   *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER J, PRT
C             В ФОРТРАНЕ IBM 360-370 АЦПУ ИМЕЕТ НОМЕР 6
      DATA PRT/6/
C
C             ВЫЗОВ ADD1 С ПАРАМЕТРОМ 1
      CALL ADD1 (1)
C
C             ИЗМЕНИЛОСЬ ЛИ ЗНАЧЕНИЕ КОНСТАНТЫ?
      J = 1
      WRITE (PRT,200) J
200   FORMAT (10X, 'J=', I1)
      STOP
      END

      SUBROUTINE ADD1 (I)
      INTEGER I
      I = I+1
      RETURN
      END
```

Рис. 2.2. Изменение константы, переданной в качестве параметра подпрограмме.

рассмотрим программу на рис. 2.2. Если бы программа была выполнена при использовании одного из вышеупомянутых диалектов, то в ячейку памяти, предназначенную для хранения константы, равной 1, было бы записано значение 2 как результат выполнения подпрограммы ADD1. Можно представить себе вашу реакцию, когда вы узнаете, что в результате выполнения инструкции J=1 J присваивается значение 2!

2.3. Дополнительные сведения о процедурах

В этом разделе мы вводим четыре нетривиальных понятия, относящихся к процедурам: альтернативные выходы из подпрограмм, дополнительные входы в подпрограмму, процедуры в качестве параметров, регулируемые размерности. Мы приведем также примеры, иллюстрирующие каждое из этих понятий. В разд. 2.4 дан пример, объединяющий все это в одной программе.

2.3.1. Альтернативные выходы из подпрограмм

Во многих диалектах Фортрана обеспечена возможность возврата из подпрограммы не к той инструкции, которая следует непосредственно за инструкцией, вызывающей подпрограмму, а к какой-либо другой. Для этого инструкция SUBROUTINE должна включать по одной звездочке (*) в списке формальных параметров для каждой метки инструкции, на которую может произойти возврат. Звездочки должны стоять перед любыми другими формальными параметрами. Хотя такой порядок расположения формальных параметров не обязателен для всех диалектов Фортрана, придерживаясь его, можно быть уверенным, что программа правильно сработает в другом диалекте, если, конечно, он обеспечивает возможность альтернативных выходов из подпрограмм.

Выполнение инструкции вида

RETURNn

где n — целая константа или переменная, приводит к возврату из подпрограммы по n-му альтернативному выходу. Если k альтернативных выходов обозначены k звездочками, то список параметров в обращении к подпрограмме должен начинаться с k меток, играющих роль параметров. Каждая метка-параметр имеет вид $\&s_n$ (или $*s_n$ в некоторых диалектах), где s_n — номер исполняемой инструкции в вызывающей программе, на которую может произойти возврат из подпрограммы.

На рис. 2.3 приведена программа, содержащая подпрограмму с двумя альтернативными выходами. Первый альтернативный выход выбирается в том случае, если список LIST отсортирован в порядке возрастания; второй — в случае, если LIST отсортирован в порядке убывания; если LIST не отсортирован, происходит нормальный возврат из подпрограммы. (*Замечание:* если список отсортирован как по возрастанию, так и по убыванию, т. е. если элементы равны между собой, то он рассматривается как отсортированный в порядке возрастания.)

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C*  ПРОГРАММА ВВОДИТ СПИСОК LIST, АНАЛИЗИРУЕТ ЕГО И ВЫДАЕТ            *
C*  СООБЩЕНИЕ, УКАЗЫВАЯ, ОТСОРТИРОВАН СПИСОК ИЛИ НЕТ                 *
C*  И В КАКОМ ПОРЯДКЕ.                                                 *
C*                                                                     *
C*  ВЫЗЫВАЕМЫЕ ПРОГРАММЫ: SORTED                                      *
C*                                                                     *
C*  ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                               *
C*    LIST — АНАЛИЗИРУЕМЫЙ СПИСОК ЦЕЛЫХ ВЕЛИЧИН,                     *
C*           ЕГО МАКСИМАЛЬНЫЙ РАЗМЕР 50.                               *
C*    N    — ЧИСЛО ЭЛЕМЕНТОВ В СПИСКЕ                                  *
C*    ПРОГРАММИСТ: ---              ДАТА: ---                          *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 2.3. Альтернативные выходы. (Текст в инстр. **4**, **11**, **21** — соответственно: "СПИСОК НЕ ОТСОРТИРОВАН", "СПИСОК ОТСОРТИРОВАН В ПОРЯДКЕ ВОЗРАСТАНИЯ", "СПИСОК ОТСОРТИРОВАН В ПОРЯДКЕ УБЫВАНИЯ".)

```
      INTEGER LIST(50), N, I, RDR, PRT
      DATA RDR/5/, PRT/6/
C
C        ВВЕСТИ АНАЛИЗИРУЕМЫЙ СПИСОК И ВЫЗВАТЬ ПОДПРОГРАММУ SORTED
C        ВОЗВРАТ НОРМАЛЬНОЙ, ЕСЛИ СПИСОК НЕ СОРТИРОВАН
C        ВОЗВРАТ НА МЕТКУ 10, ЕСЛИ СПИСОК СОРТИРОВАН В ВОСХОДЯЩЕМ
C        ПОРЯДКЕ, И НА МЕТКУ 20, ЕСЛИ В НИСХОДЯЩЕМ ПОРЯДКЕ
      READ (RDR,2) N
2     FORMAT (I2)
      IF ((N .LE. 0) .OR. (N .GT. 50)) STOP
      READ (RDR,3) (LIST(I), I = 1, N)
3     FORMAT (5I10)
      CALL SORTED (&10, &20, LIST, N)
      WRITE (PRT,4)
4     FORMAT ('
      STOP
10    WRITE (PRT,11)
11    FORMAT ('
      STOP
20    WRITE (PRT,21)
21    FORMAT ('
      STOP
      END

      SOBROUTINE SORTED (*,*, LIST, N)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ПОДПРОГРАММА SORTED (*, *, LIST, N)                                  *
C* НАЗНАЧЕНИЕ: ПРОВЕРЯЕТ, ОТСОРТИРОВАНЫ ЛИ ЭЛЕМЕНТЫ СПИСКА,            *
C*             ЕСЛИ НЕТ, НОРМАЛЬНЫЙ ВОЗВРАТ                             *
C*             ЕСЛИ В ВОЗРАСТАЮЩЕМ ПОРЯДКЕ, ВОЗВРАТ RETURN1             *
C*             ЕСЛИ ВСЕ ЭЛЕМЕНТЫ СПИСКА РАВНЫ, ВОЗВРАТ RETURN1          *
C*             ЕСЛИ В НИСХОДЯЩЕМ ПОРЯДКЕ, ВОЗВРАТ RETURN 2              *
C*                                                                     *
C*                                                                     *
C* ПАРАМЕТРЫ: LIST — ЦЕЛЫЙ МАССИВ, РАЗМЕР 50                           *
C*            N — КОЛИЧЕСТВО ЭЛЕМЕНТОВ СПИСКА                           *
C* ПОБОЧНЫЕ ЭФФЕКТЫ: НЕТ                                                *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER LIST(50), N, I
      LOGICAL ASCEND, DESCND
C
C        СПИСОК ИЗ 1 ЭЛЕМЕНТА ОТСОРТИРОВАН
      IF (N .LE. 1) RETURN1
C
C        ПРЕДПОЛОЖИМ, ЧТО СПИСОК ОТСОРТИРОВАН В ВОСХОДЯЩЕМ
C        И НИСХОДЯЩЕМ ПОРЯДКЕ. ИЗМЕНИМ ЗНАЧ. ЛОГИЧЕСКОЙ
C        ПЕРЕМЕННОЙ, ЕСЛИ ВСТРЕТИМ „НЕУПОРЯДОЧЕННУЮ" ПАРУ
      ASCEND = .TRUE.
      DESCND = .TRUE.
C
C        ИССЛЕДУЕМ КАЖДУЮ ПАРУ: ИЩЕМ „НЕУПОРЯДОЧЕННЫЕ" ЭЛЕМЕНТЫ
      DO 10 I = 2, N
         IF (LIST(I-1) .GT. LIST(I)) ASCEND = .FALSE.
         IF (LIST(I-1) .LT. LIST(I)) DESCND = .FALSE.
10    CONTINUE
C
C        ВОЗВРАТ, ОСНОВАННЫЙ НА РЕЗУЛЬТАТЕ ПРОСМОТРА ПАР СПИСКА
      IF (ASCEND) RETURN1
      IF (DESCND) RETURN2
      RETURN
      END
```

Продолжение Рис. 2.3.

2.3.2. Дополнительные входы в процедуру

Часто возникает такая ситуация, когда для двух или более близких по смыслу вычислений необходимо создать процедуры. Например, для аппроксимации синуса, косинуса и многих других тригонометрических функций можно применить, по существу, один и тот же алгоритм; различаются только значения некоторых начальных коэффициентов.

При нормальном входе в подпрограмму ее выполнение начинается с первой исполняемой инструкции. Во многих вариантах Фортрана имеется инструкция ENTRY, которая позволяет начать выполнение с других точек подпрограммы. Инструкция ENTRY имеет следующий вид:

ENTRY *имя* [*(список-форм-парам)*]

Если такая инструкция присутствует в подпрограмме, то при вызове с указанием имени входа выполнение подпрограммы начинается с первой исполняемой инструкции, следующей за этим входом. Если инструкция ENTRY находится в функции и в обращении указано *имя* входа, то выполнение этой функции начинается с первой исполняемой инструкции, следующей за ENTRY. Хотя это строго и не требуется, списки формальных параметров во всех инструкциях ENTRY должны быть такими же, как в соответствующих инструкциях SUBROUTINE или FUNCTION. Кроме того, *имя* в ENTRY должно иметь тот же тип, что и *имя* в соответствующей инструкции FUNCTION. При несоблюдении этих рекомендаций программа может давать правильный результат в одних диалектах Фортрана и ошибочный в других.

```
      INTEGER FUNCTION LOOK1 (LIST, N, ARG)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C*  ФУНКЦИЯ ПОИСКА В ТАБЛИЦЕ. ЭТА ФУНКЦИЯ ВЫПОЛНЯЕТ ПОИСК *
C*  ЗНАЧЕНИЯ 'ARG' В ТАБЛИЦЕ 'LIST'. ЕСЛИ 'LIST' ОТСОРТИРОВАНА, *
C*  ТО ВХОД-ПО ИМЕНИ LOOK2; ЕСЛИ НЕ ОТСОРТИРОВАНА ,ТО —        *
C*  ПО ИМЕНИ 'LOOK1'. В ПОСЛЕДНЕМ СЛУЧАЕ 'LIST' СОРТИРУЕТСЯ    *
C*  ПОДПРОГРАММОЙ 'SORT'. ЕСЛИ 'ARG' НАЙДЕНО В 'LIST(K)'; ТО   *
C*  ВОЗВРАЩАЕТСЯ ЗНАЧЕНИЕ K. ЕСЛИ 'ARG' НЕ НАЙДЕНО,            *
C*  ВОЗВРАЩАЕТСЯ НОЛЬ.                                          *
C*                                                              *
C*  ПАРАМЕТРЫ:                                                  *
C*    LIST — ТАБЛИЦА МОЖЕТ БЫТЬ ОТСОРТИРОВАНА ИЛИ               *
C*           НЕ ОТСОРТИРОВАНА, ТИП ЦЕЛЫЙ, РАЗМЕР 50;           *
C*       N — ЧИСЛО ЭЛЕМЕНТОВ В ТАБЛИЦЕ 'LIST';                  *
C*  ARG    — ЦЕЛОЕ ЗНАЧЕНИЕ, КОТОРОЕ ИЩЕТСЯ В 'LIST'.          *
C*                                                              *
C*  ТОЧКИ ВХОДА:                                                *
C*    LOOK1 — ИСПОЛЬЗУЕТСЯ, ЕСЛИ 'LIST' НЕ ОТСОРТИРОВАНА;      *
C*    LOOK2 — ИСПОЛЬЗУЕТСЯ, ЕСЛИ 'LIST' ОТСОРТИРОВАНА          *
C*            В ПОРЯДКЕ ВОЗРАСТАНИЯ.                            *
C*  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                   *
C*    SORT(LIST,N) — СОРТИРУЕТ N ЭЛЕМЕНТОВ 'LIST' В ПОРЯДКЕ     *
C*                   ВОЗРАСТАНИЯ ЗНАЧЕНИЙ.                      *
C*                   ** ПРИМЕЧАНИЕ ** ПОДПРОГРАММА SORT ЗДЕСЬ   *
C*                   НЕ ПРИВОДИТСЯ.                             *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 2.4. Пример дополнительных входов в процедуру.

```
      INTEGER LIST(50), N, ARG, LOOK2
C
C     LIST НЕОТСОРТИРОВАН. ОТСОРТИРОВАТЬ
      CALL SORT (LIST, N)
C
C     ЗДЕСЬ ВХОД ДЛЯ ОТСОРТИРОВАН. LIST
      ENTRY LOOK2 (LIST, N, ARG)
C
C     СОДЕРЖИТСЯ ЛИ 'ARG' В 'LIST'
      DO 10 LOOK1 = 1, N
         IF (LIST(LOOK1) .GE. ARG) GO TO 20
10    CONTINUE
C
C     ПЕРЕХОД—В 'LIST' НЕ СОДЕРЖИТСЯ
      GO TO 30
C
C     ЕСЛИ 'ARG' ЕСТЬ, ТО В ПОЗИЦИИ LOOK1
20    IF (LIST(LOOK1) .EQ. ARG) RETURN
C
C     'ARG' ОТСУТСТВУЕТ, ВЕРНУТЬ НОЛЬ
30    LOOK1 = 0
      RETURN
      END
```

Продолжение Рис. 2.4.

На рис. 2.4 приведен пример функции с двумя входами. Вход через имя LOOK1 приводит к сортировке списка LIST и затем к поиску ARG в LIST. Вход через точку LOOK2 предполагает, что таблица LIST уже отсортирована, и в этом случае осуществляется только поиск ARG. В обоих случаях функция возвращает наименьшее значение K, такое, что ARG=LIST(K), и ноль, если ARG не найден. Заметим, что мы присваиваем результирующее значение только переменной LOOK1, поскольку все имена входов в подпрограмму-функцию ассоциируются с одной и той же областью памяти. В этом смысле имя LOOK1 является синонимом имени LOOK2 в теле функции.

2.3.3. Процедуры в роли параметров

Имена функций или подпрограмм могут появиться в списке параметров других процедур. В этом случае чаще всего просто необходимо и всегда желательно записать инструкцию EXTERNAL. Она имеет следующий вид:

EXTERNAL *список имен*

Все имена процедур, используемые в качестве параметров, должны быть перечислены в инструкции EXTERNAL. Любое имя процедуры, встретившееся в списке параметров, должно быть определено соответствующим образом. Если это не сделано, компилятор Фортрана будет рассматривать имя как простую переменную.

Рассмотрим задачу приближенного вычисления площади под кривой f в интервале от точки a до точки b. Простой способ решения этой задачи называется *методом трапеций*. Согласно этому методу, площадь вычисляется так:

$$(b-a)*(f(a)+f(b))/2$$

Это действительно площадь трапеции со сторонами $(a, 0)$, $(a, f(a))$, $(b, f(b))$, $(b, 0)$. (См. левую часть рис. 2.5.) Лучшее приближение получается делением интервала $[a, b]$ на n подынтервалов одинаковой длины:

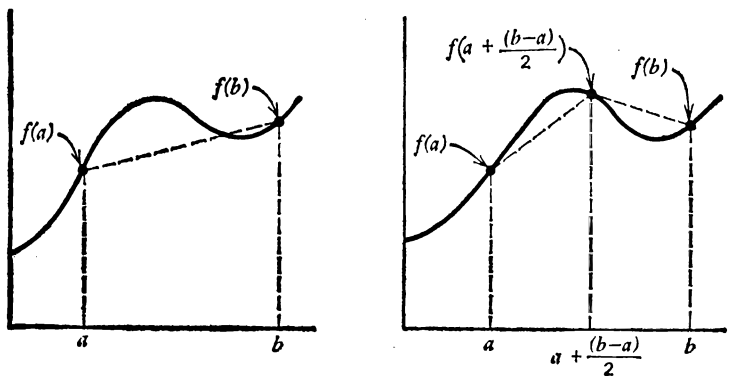


Рис. 2.5. Правило трапеций.

$[a, a+(b-a)/n]$, $[a+(b-a)/n, a+2*(b-a)/n]$, . . .,$[b-(b-a)/n, b]$
Тогда площадь в любом подынтервале $[c, d]$ будет приближенно равна

$$(d-c)*(f(c)+f(d))/2$$

Сумма всех n таких площадей дает аппроксимацию площади в интервале от a до b (см. правую часть рис. 2.5).

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C*  ПРОГРАММА ВЫЧИСЛЕНИЯ ПЛОЩАДИ КРИВОЙ X ** 2-X                    *
C*  В ИНТЕРВАЛЕ МЕЖДУ 1 И 4. ПЛОЩАДЬ ВЫЧИСЛЯЕТСЯ                    *
C*  ПРИБЛИЖЕННО ПО МЕТОДУ ТРАПЕЦИЙ.                                  *
C*  ИНТЕРВАЛ РАЗБИТ НА 12 ПОДЫНТЕРВАЛОВ.                             *
C*                                                                   *
C*  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                         *
C*    AREA(A, B, N, F)-ИСПОЛЬЗУЕТ МЕТОД ТРАПЕЦИЙ ДЛЯ                *
C*                     ВЫЧИСЛЕНИЯ ПЛОЩАДИ ПОД КРИВОЙ F             *
C*                     В ИНТЕРВАЛЕ ОТ A ДО B. N ОПРЕДЕЛЯЕТ ЧИСЛО    *
C*                     ПОДЫНТЕРВАЛОВ.                               *
C*
C*  НЕОБЫЧНЫЕ СИТУАЦИИ:
C*    ФУНКЦИЯ FCN ПЕРЕДАЕТСЯ В КАЧЕСТВЕ ПАРАМЕТРА                   *
C*    В ПОДПРОГРАММУ AREA. ФУНКЦИЯ FCN. ВЫЧИСЛЯЕТ X ** 2-X.         *
C*    ПРОГРАММИСТ: - - -          ДАТА  - - -                      *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 2.6. Интегрирование по правилу трапеций. (Текст в инстр. **200**: "ПЛОЩАДЬ ПОД X** 2—X МЕЖДУ 1 и 4".)

```
      INTEGER PRT
      REAL APPROX, AREA
      DATA PRT/6/
C
C             НУЖНА ИНСТРУКЦИЯ EXTERNAL, ИНФОРМИРУЮЩАЯ КОМПИЛЯТОР,
C             ЧТО FCN ЭТО ПОДПРОГРАММА, А НЕ ПЕРЕМЕННАЯ
      EXTERNAL FCN
C
C             ВЫЧИСЛЕНИЕ ПЛОЩАДИ ПОД FCN В ИНТЕРВАЛЕ ОТ 1 ДО 4
      APPROX = AREA(1., 4., 12, FCN)
      WRITE (PRT,200) APPROX
200   FORMAT('0', 5X, 'THE AREA UNDER X**2-X BETWEEN 1 AND 4 IS', F10.2)
      STOP
      END


      REAL FUNCTION FCN (X)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВЫЧИСЛЕНИЕ X**2 - X. ЕСЛИ НУЖНО ВЫЧИСЛИТЬ ПЛОЩАДЬ ПОД              *
C* БОЛЕЕ СЛОЖНОЙ ФУНКЦИЕЙ, ИЗМЕНИТЕ ЭТУ ПОДПРОГРАММУ                  *
C* ВНЕСИТЕ ТАКЖЕ ИЗМЕНЕНИЯ В ГЛАВНУЮ ПРОГРАММУ,                        *
C* ЕСЛИ ИЗМЕНИТСЯ ИНТЕРВАЛ ИНТЕГРИРОВАНИЯ.                             *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      REAL X
      FCN = X**2 - X
      RETURN
      END

      REAL FUNCTION AREA (A, B, N, F)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВЫЧИСЛЕНИЕ ПЛОЩАДИ ПОД F ПО ПРАВИЛУ ТРАПЕЦИЙ                        *
C*                                                                      *
C* ПАРАМЕТРЫ:                                                           *
C*     A - НИЖНЯЯ ГРАНИЦА ИНТЕРВАЛА (ВЕЩЕСТВЕННОЕ ЗНАЧЕНИЕ),           *
C*     B - ВЕРХНЯЯ ГРАНИЦА ИНТЕРВАЛА (ВЕЩЕСТВЕННОЕ ЗНАЧЕНИЕ),          *
C*     N - ЧИСЛО ПОДИНТЕРВАЛОВ (ЦЕЛОЕ ЗНАЧЕНИЕ),                       *
C*     F - ВЕЩЕСТВЕННАЯ ФУНКЦИЯ.                                        *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER N, I
      REAL A, B, F, INTRVL, C, D
C
C             ДЕЛИМ АВ НА ПОДИНТЕРВАЛЫ ДЛИНОЙ (B-A)/N.
C             ВЫЧИСЛИМ ПЛОЩАДЬ ТРАПЕЦИЙ НА КАЖДОМ ИЗ НИХ.
C             СУММИРУЕМ ПЛОЩАДИ ВСЕХ ТРАПЕЦИЙ.
      AREA = 0.
      INTRVL = (B-A) / N
      DO 10 I = 1, N
         C = A + INTRVL*(I-1)
         D = A + INTRVL*I
         AREA = AREA + INTRVL*(F(C) + F(D))/2.
10    CONTINUE
      RETURN
      END
```

Продолжение Рис. 2.6.

На рис. 2.6 представлена процедура AREA с четырьмя параметрами: A, B, N и F. A и B имеют вещественные значения, представляющие соответственно начало и конец интервала, на котором выполняется аппроксимация. N — это число подынтервалов, а F — формальный параметр, используемый в качестве имени функции. Фактический параметр, связанный с F, должен быть именем функции вещественного типа. Эта функция определяет кривую, площадь под которой мы вычисляем.

2.3.4. Переменные размерности

Написанные нами подпрограммы сортировки и поиска имеют общее назначение. В этих подпрограммах список, подлежащий сортировке и поиску, задан в качестве параметра. Однако применение подпрограмм ограничивается фиксированными размерами списка. Например, подпрограмма, приведенная на рис. 2.4, предназначена для поиска в таблицах, размер которых не превышает 50 элементов.

Формальные параметры-массивы могут иметь переменные, а не фиксированные размеры. Чтобы их задать таким образом, размер массива также должен быть указан в качестве параметра. Тогда или декларация DIMENSION, или декларация типа принимает следующую форму:

```
DIMENSION LIST1(N), LIST2 (N, M)
```

где LIST1, LIST2, N и M — формальные параметры. Данная декларация объявляет, что LIST1 имеет размер N, а LIST2— размер N*M. Поскольку массивы всегда ассоциированы по адресу, никакой области памяти для LIST1 и LIST2 фактически не отводится. DIMENSION только объявляет LIST1 и LIST2 массивами, ограничивает допустимые диапазоны изменения индексов и позволяет вычислить позицию, занимаемую в памяти каждым элементом. При использовании данного метода фактические параметры должны иметь размеры соответственно не меньше, чем N и N*M.

2.4. Поиск в таблицах при использовании хеширования

В гл. 1 мы познакомили вас с двумя методами организации таблиц и поиска в таблицах из n элементов. Первый метод основан на последовательном (линейном) поиске и требует в среднем порядка n сравнений для нахождения элемента. Второй метод основан на бинарном поиске. Он использует упорядоченный список и требует только примерно $\log_2(n)$ сравнений. Недостатком бинарного поиска является то, что он требует упорядоченной таблицы. Так как общие алгоритмы сортировки требуют в среднем выполнения по крайней мере $n*\log_2(n)$ шагов, их применение сопряжено с довольно серьезными затратами, особенно если таблица при работе имеет тенденцию к увеличению или если число поисков мало.

Третий общепринятый метод организации таблиц и поиска в таблицах называется *хешированием*. Время, необходимое для вставки или нахождения элемента в таблице при использовании этого метода, зависит от степени заполнения списка на данный

момент и не зависит от размера списка. Приведенный ниже пример показывает зависимость среднего числа проб (т. е. сравнений при поиске) от процента заполнения хеш-таблицы:

Процент заполнения таблицы	Среднее число проб
10	1.06
50	1.50
75	2.50
90	5.50

Метод хеширования заключается в следующем. Мы выбираем функцию h, которая отображает каждый элемент таблицы (например, номер в телефонном справочнике) в последовательность цифр, задающих позицию этого элемента в памяти. Функция

$$h(k)=k-(k/100)*100+1$$

например, ставит в соответствие любому k число в диапазоне от 1 до 100. Характерно, что если последние две цифры числа k представляют собой число q, то $h(k)=q+1$. Таким образом, если последние две цифры значений k равномерно распределены между 00 и 99, то значения $h(k)$ будут равномерно распределены между 1 и 100.

Идеальной хеш-функцией является такая функция, которая для каждого имеющегося или потенциального элемента данных задает единственную позицию в таблице. Этот элемент затем запоминается или разыскивается в своей позиции в соответствии с его хеш-кодом. Например, хеш-функция h, приведенная выше, была бы идеальной, если бы значения k находились в диапазоне между 0 и 99. Если, однако, мы имеем дело с девятью значащими цифрами (например, с номерами социального страхования), то для идеальной хеш-функции потребовалась бы таблица с размером в миллиард слов. Ясно, что в общем случае диапазон значений больше, чем число машинных слов, по которым мы хотим рассеять элементы таблицы. Вследствие этого мы не можем с гарантией отвести свободное слово в таблице для каждого потенциального значения элемента данных. Таким образом, возникает *коллизия:* мы хотим вставить в таблицу величину k, а находим, что позиция $h(k)$ уже занята. При возникновении коллизии мы должны найти другую позицию для величины k. Частота, с которой возникают коллизии, и количество проб, необходимых для нахождения «вторичного» свободного места в таблице, как раз и определяют степень сложности хеширования.

Для разрешения коллизий разработано большое количество различных способов. Здесь мы сконцентрируем наше внимание на самом простом способе, *линейном опробовании.* Если требуется вставить элемент k, то мы анализируем позицию $h(k)$. Если позиция занята, пытаемся найти свободное место в позиции

$h(k)+1$ [или в 1 при $h(k)=n$]. В случае неудачи процесс продолжается для $h(k)+2$, $h(k)+3$,... до тех пор, пока не будет найдена свободная позиция или выяснится, что k уже содержится в таблице или что таблица уже заполнена до отказа.

Следующий пример показывает весьма общее описание процесса инициализации, построения и поиска в хеш-таблице. Таблица состоит из n элементов; k — это элемент, который нужно вставить или найти в таблице; h — хеш-функция; все значения k больше нуля.

Инициализация: /Очистить все позиции в таблице/

$table(i) \leftarrow -1$, $i=1$, n /Все позиции свободны/

Поиск: /Посмотреть, находится ли в таблице значение ключа k/

$p \leftarrow h(k)$ /Начальная проба в хеш-позиции/
$s \leftarrow p$ /Запомнить начальную точку/

Цикл 1: /Проверить p-ю позицию. При неудаче проверить следующую/
если $table(p)=k$, то возврат ('поиск завершен удачно в позиции k')/k найдено?/
если $table(p)=-1$, то возврат ('неудача') /Найдена свободная позиция?/
$p \leftarrow p+1$ /Приращение для перехода к следующей позиции/
если $p>n$, то $p \leftarrow 1$ /Таблица рассматривается как круговая/
если $p=s$, то возврат ('неудача') /Обратно к начальной точке?/
перейти на цикл 1 /Проверить следующую позицию/

Вставка: /Вставить значение ключа k/

$p \leftarrow h(k)$ /Начальная проба в хеш-позиции/
$s \leftarrow p$ /Запомнить начальную точку/

Цикл 2: /Проверить p-ю позицию. При неудаче проверить следующую/
если $table(p)=k$, то возврат ('такой ключ уже есть') /Найден?/
если $table(p)=-1$, то /Свободное место найдено?/
начало /Вставить ключ в свободную позицию/
$table(p)=k$
возврат ('успех')
конец
$p \leftarrow p+1$ /Приращение для перехода к следующей позиции/
если $p>n$, то $p \leftarrow 1$ /Таблица рассматривается как круговая/
если $p=s$, то возврат ('переполнение') /Таблица полна?/
перейти на цикл2 /Проверить следующую позицию/

На рис. 2.7 приведен набор подпрограмм, которые могут быть использованы для реализации изложенных алгоритмов работы с хеш-таблицами. Подпрограмма INIT выполняет инициализацию таблицы. Подпрограмма HASH имеет две точки входа LOOKUP и INSERT. Одна точка входа служит для организации поиска в таблице, вторая — для вставки данных. Параметр TABLE представляет хеш-таблицу, K — это искомый или вставляемый элемент и H — хеш-функция. Если вход в подпрограмму HASH происходит через INSERT, то выполняются следующие операции:

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИСПОЛЬЗОВАНИЕ ХЕШ-ФУНКЦИИ ПРИ ОРГАНИЗАЦИИ ТАБЛИЦЫ И ПОИСКЕ.          *
C* ПРОГРАММА ИЛЛЮСТРИРУЕТ РАЗЛИЧНЫЕ ТОЧКИ ВХОДА,                        *
C* АЛЬТЕРНАТИВНЫЕ ВЫХОДЫ И РЕГУЛИРУЕМЫЕ РАЗМЕРНОСТИ                      *
C*                                                                       *
C* ВВОД:                                                                 *
C*   КАРТА, ОПРЕДЕЛЯЮЩАЯ КОЛИЧЕСТВО ВСТАВОК ( ФОРМАТ I3) ;              *
C*   ПО ОДНОЙ КАРТЕ НА ВСТАВЛЯЕМОЕ ЗНАЧЕНИЕ ( ФОРМАТ I6 ) ;             *
C*   ПО ОДНОЙ КАРТЕ НА ЧИСЛО, КОТОРОЕ ИЩЕТСЯ В ТАБЛИЦЕ ( ФОРМАТ I6 )    *
C*                                                                       *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                             *
C*    INIT, INSERT, LOOKUP                                               *
C*                                                                       *
C* ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                                  *
C*   TABLE - ТАБЛИЦА, В КОТОРУЮ ПОМЕЩАЮТСЯ ЗНАЧЕНИЯ С ПОМОЩЬЮ           *
C*           ХЕШ-КОДА. ТИП - ЦЕЛЫЙ, РАЗМЕР 100;                         *
C*   ITEMS - КОЛИЧЕСТВО ЦЕЛЫХ ЗНАЧЕНИЙ В ТАБЛИЦЕ (ТИП ЦЕЛЫЙ)            *
C*                                                                       *
C* ОГРАНИЧЕНИЯ:                                                          *
C*   В ТАБЛИЦУ МОГУТ БЫТЬ ПОМЕЩЕНЫ ТОЛЬКО ПОЛОЖИТЕЛЬНЫЕ ЧИСЛА.           *
C*   ОТРИЦАТЕЛЬНЫЕ ВЕЛИЧИНЫ ОЗНАЧАЮТ СВОБОДНУЮ ОБЛАСТЬ                  *
C*                                                                       *
C* ПРОГРАММИСТ: ---                              ДАТА: ---               *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER TABLE(100), K, POS, ITEMS, RDR, PRT
      DATA RDR/5/, PRT/6/
      EXTERNAL H
C.
C     СНАЧАЛА ВСЕ ЭЛЕМЕНТЫ ТАБЛИЦЫ РАВНЫ -1 (ТАБЛИЦА ПУСТА).
      CALL INIT (TABLE, 100)
C

C       ВВОД КОЛИЧЕСТВА ЭЛЕМЕНТОВ, ЗАТЕМ ЗАПОМИНАЕМЫХ ЗНАЧЕНИЙ.
C       ВЫВОД СООБЩЕНИЯ ПРИ ПОЯВЛЕНИИ ДУБЛИРОВАННЫХ ЗНАЧЕНИЙ.
C       СООБЩЕНИЕ ОБ ОШИБКЕ И ОСТАНОВ ПРИ ПЕРЕПОЛНЕНИИ ТАБЛИЦЫ
      READ (RDR,2) ITEMS
2     FORMAT (I3)
      DO 10 I = 1, ITEMS
         READ (RDR,6) K
6        FORMAT (I6)
C
C       УДАЧНАЯ ВСТАВКА - ПЕРЕХОД НА 10, ПЕРЕПОЛНЕНИЕ - НА 888.
C       НОРМАЛЬНЫЙ ВОЗВРАТ ПРИ ДУБЛИРОВАНИИ ЗНАЧЕНИЯ
         CALL INSERT (&10, &888, TABLE, 100, K, POS, H)
         WRITE (PRT,7) K, POS
7        FORMAT (' DUPLICATE ITEM', I7, ' FOUND AT POSITION', I4)
10    CONTINUE
C
C       ВЫВОД ТЕКУЩЕГО СОДЕРЖИМОГО ТАБЛИЦЫ
      WRITE (PRT,15) TABLE
15    FORMAT ('0',T30, 'HASH TABLE AFTER ALL INSERTIONS' // ( 10I8) )
C
C       ВВОД РАЗЫСКИВАЕМЫХ ЗНАЧЕНИЙ ДО КОНЦА ФАЙЛА.
C       ДЛЯ КАЖДОГО ЗНАЧЕНИЯ К СМОТРИМ, ЕСТЬ ЛИ ОНО В ТАБЛИЦЕ.
C       ЕСЛИ ДА, ТО ГДЕ НАХОДИТСЯ. ЕСЛИ НЕТ, СООБЩЕНИЕ
20    READ (RDR,6,END=999) K
C
C       ЕСЛИ К В ТАБЛИЦЕ НЕТ, ПЕРЕХОД НА 25, ИНАЧЕ ПРОДОЛЖЕНИЕ
      CALL LOOKUP (&25, &25, TABLE, 100, K, POS, H)
         WRITE (PRT,22) K, POS
22       FORMAT (' ITEM', I7, ' FOUND AT POSITION', I4)
      GO TO 20
25       WRITE (PRT,26) K
26       FORMAT (' ITEM', I7, ' NOT FOUND')
      GO TO 20
C
C       ВЫХОДИМ СЮДА ПРИ ПЕРЕПОЛНЕНИИ
888   WRITE (PRT,889)
889   FORMAT (' *** TABLE OVERFLOW ***')
999   STOP
      END
```

Рис. 2.7. Организация хеш-таблицы.

```
      SUBROUTINE INIT (TABLE, N)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* СНАЧАЛА ВСЕ N ЭЛЕМЕНТОВ СПИСКА 'TABLE' ПРИРАВНИВАЮТСЯ -1.          *
C* ЭТО ОЗНАЧАЕТ, ЧТО ПОЗИЦИИ СВОБОДНЫ                                  *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER N, TABLE(N), I, EMPTY/-1/
      DO 10 I = 1, N
         TABLE(I) = EMPTY
10    CONTINUE
      RETURN
      END

      INTEGER FUNCTION H (K)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ХЕШ-ФУНКЦИЯ: ПО ДАННОМУ K ВОЗВРАЩАЕТ 2-РАЗРЯДНОЕ ЧИСЛО            *
C* ИЗ 2 КРАЙНИХ ПРАВЫХ ЦИФР K ПЛЮС 1                                  *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER K
      H = K - (K/100)*100 + 1
      RETURN
      END

      SUBROUTINE HASH (*, *, TABLE, N, K, POS, H)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВСТАВИТЬ ЗНАЧЕНИЕ K В 'TABLE', ИСПОЛЬЗУЯ ХЕШ-ФУНКЦИЮ 'H'.          *
C* ЕСЛИ 'TABLE' НЕ ЗАПОЛНЕНА И 'K' В НЕЙ НЕТ, ТО RETURN1 ПОСЛЕ ВСТАВКИ K *
C* (ЕСЛИ ОБРАЩЕНИЕ ВЫПОЛНЯЛОСЬ С ЭТОЙ ЦЕЛЬЮ).                          *
C* ЕСЛИ 'TABLE' ЗАПОЛНЕНА И 'K' В НЕЙ НЕТ, ТО RETURN2.                  *
C* ЕСЛИ 'K' УЖЕ ЕСТЬ В ТАБЛИЦЕ, ТО RETURN.                             *
C*                                                                      *
C* ПАРАМЕТРЫ:                                                           *
C*    TABLE - ХЕШ-ТАБЛИЦА; ТИП ЦЕЛЫЙ; РАЗМЕР N                          *
C*    N     - РАЗМЕР ТАБЛИЦЫ; ТИП ЦЕЛЫЙ                                 *
C*    K     - КЛЮЧ ДЛЯ ВСТАВКИ ИЛИ ПОИСКА; ТИП ЦЕЛЫЙ                   *
C*    POS   - ПОЗИЦИЯ, КУДА ВСТАВЛЕНО ИЛИ ГДЕ НАЙДЕНО K; ТИП ЦЕЛЫЙ     *
C*    H     - ПОДПРОГРАММА ХЕШ-ФУНКЦИИ; ТИП ЦЕЛЫЙ                      *
C*                                                                      *
C* ТОЧКИ ВХОДА:                                                         *
C*   INSERT - ДЛЯ ВСТАВКИ K В ТАБЛИЦУ                                   *
C*   LOOKUP - ДЛЯ ПОИСКА K В ТАБЛИЦЕ                                    *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      INTEGER N, TABLE(N), K, POS, H, S, EMPTY/-1/
      LOGICAL PUTIN
C
C        ВХОД ДЛЯ ВСТАВКИ K В ТАБЛИЦУ; PUTIN <=.TRUE.,
C        ЧТОБЫ УКАЗАТЬ, ЧТО НУЖНО ВСТАВИТЬ ЗНАЧЕНИЕ
      ENTRY INSERT (*, *, TABLE, N, K, POS, H)
      PUTIN = .TRUE.
      GO TO 10
C
C        ВХОД ДЛЯ ПОИСКА K В ТАБЛИЦЕ. PUTIN <=.FALSE.,
C        ЧТОБЫ УКАЗАТЬ, ЧТО ВСТАВЛЯТЬ K НЕ НУЖНО
      ENTRY LOOKUP (*, *, TABLE, N, K, POS, H)
      PUTIN = .FALSE.
C
C        ОБЩАЯ СЕКЦИЯ. ЗДЕСЬ СХОДЯТСЯ ОБЕ ВЕТВИ
C
C        ВЫЧИСЛИТЬ ХЕШ-ПОЗИЦИЮ ДЛЯ K. ЗАПОМНИТЬ ЕЕ
C        ДЛЯ ФАЗЫ ЛИНЕЙНОГО ОПРОБОВАНИЯ
10    POS = H(K)
      S = POS
C
C        ЛИНЕЙНОЕ ОПРОБОВАНИЕ - НАЙТИ СВОБОДНУЮ ПОЗИЦИЮ
C        ДЛЯ K ИЛИ УСТАНОВИТЬ, ЧТО K УЖЕ В ТАБЛИЦЕ.
C        ВЫХОД НА 30, ЕСЛИ МЕСТО НАЙДЕНО. RETURN, ЕСЛИ K НАЙДЕНО;
C        RETURN2, ЕСЛИ ТАБЛИЦА ЗАПОЛНЕНА И K В НЕЙ НЕТ
20    IF (TABLE(POS) .EQ. K) RETURN
         IF (TABLE(POS) .EQ. EMPTY) GO TO 30
C
C        ТАБЛИЦА РАССМАТРИВАЕТСЯ КАК ЦИКЛИЧЕСКАЯ
         POS = POS + 1
         IF (POS .GT. N) POS=1
C
C        ЕСЛИ ВЕРНУЛИСЬ К ТОЧКЕ СТАРТА, ВЫЙТИ ИЗ ПРОЦЕДУРЫ
         IF (POS .EQ. S) RETURN2
      GO TO 20
C
C        НАЙДЕНО МЕСТО ДЛЯ K; ВСТАВИТЬ K, ЕСЛИ ТРЕБУЕТСЯ
30    IF (PUTIN) TABLE(POS)=K
      RETURN1
      END
```

Продолжение Рис. 2.7.

1. Если K уже имеется, скажем, в позиции TABLE(P), то происходит нормальный возврат, а POS присваивается значение P.

2. Если K не найден в таблице и TABLE (P) — первая встретившаяся свободная позиция, то TABLE(P) получает значение K и выбирается первый альтернативный возврат RETURN1.

3. Если K не найден и таблица полна, выбирается второй альтернативный возврат RETURN2.

Если вход в подпрограмму HASH происходит через LOOKUP, то предпринимаются точно такие же действия, как и для INSERT, кроме пункта 2. Здесь выход происходит через первый альтернативный возврат RETURN1, но K не вставляется в таблицу.

2.5. Инструкции DATA в подпрограммах

В гл. 0 мы обсудили применение инструкций DATA и деклараций типа для инициализации значений переменных, массивов и элементов массивов перед выполнением программы. Заметим, что при инициализации данных внутри подпрограммы можно не заботиться о начальных значениях формальных параметров. Это в общем-то понятно, поскольку формальный параметр получает значение только в результате ассоциации с фактическим параметром. Поэтому начальное значение будет проигнорировано. Более того, если передача параметров организована по ссылке, то формальным параметрам не будет отведена память, и поэтому невозможно сохранить никаких начальных данных.

2.6. COMMON — инструкция Фортрана для объявления данных глобальными

2.6.1. Локальные и глобальные данные

За исключением формальных параметров, элементы данных (переменные и массивы), используемые внутри программного модуля (процедуры или главной программы), рассматриваются как локальные в этом модуле. Это означает, что переменная с именем A в любом независимо оттранслированном модуле отлична от переменной A (или любой другой переменной), на которую происходит ссылка внутри другого модуля. Такая локализация имеет два основных достоинства.

Во-первых, она позволяет двум или более программистам работать независимо над различными модулями некоторой большой программы. Каждый программист может выбирать имена переменных, не беспокоясь, что они могут быть использованы другими программистами.

Во-вторых, уменьшается вероятность непреднамеренных побочных эффектов при взаимодействии между подпрограммами Фортрана. Если внутри подпрограммы SUB используются данные, которые либо имеют локальные имена, либо являются формальными параметрами, то в результате вызова и выполнения подпрограммы SUB в вызывающей программе могут измениться только значения фактических параметров, которые были переданы подпрограмме. Это облегчает отладку программ по сравнению с ситуацией, когда подпрограмма может модифицировать, помимо значений фактических параметров, еще и значения каких-то других переменных вызывающей программы. Кроме того, программу легче изучать. Фактически можно обнаружить побочные эффекты подпрограммы, не имея даже ее полного текста.

Однако возникают такие ситуации, когда проявляются недостатки подразумеваемого по умолчанию локального статуса данных. Если большое количество программных модулей используют один и тот же набор переменных и массивов или если для каких-либо модулей необходимо задать большое число параметров, то списки параметров могут стать очень длинными, что естественно может привести к ошибкам. Поэтому в Фортране предусмотрены также глобальные переменные, т. е. переменные, общие для нескольких модулей. Для этой цели служит инструкция COMMON, к определению и обсуждению которой мы теперь обратимся.

2.6.2. Стандартное применение инструкции COMMON

Перед выполнением программы на Фортране для каждого ее модуля отводится память. Память подразделяется на две отдельные части. Первая часть содержит все машинные команды программы, вторая часть отведена под локальные переменные и константы.

Кроме того, отдельный блок памяти отводится для использования всеми модулями. Этот блок называется COMMON (общим блоком). Для того чтобы один модуль с помощью общего блока имел возможность разделять данные с другими модулями, среди его деклараций должна быть инструкция COMMON. Стандартная форма такой инструкции:

COMMON *имя*$_1$, *имя*$_2$, . . ., *имя*$_n$

В инструкции COMMON декларируется, что применительно к данному модулю первая ячейка в общем блоке отводится под *имя*$_1$, следующая — под *имя*$_2$ и т. д.

Если инструкция COMMON слишком длинна, чтобы ее можно было уместить на одной перфокарте, она может быть продолжена по обычным правилам Фортрана или возобновлена посредством новой инструкции COMMON. Так, инструкция

```
COMMON A, B, C
```

равносильна инструкции

```
      COMMON A, B
     *       C
```

или

```
COMMON A
COMMON B, C
```

и т. п.

В качестве первого примера использования декларации COMMON рассмотрим следующие инструкции:

```
INTEGER DOLLAR, CENT, LIST(2, 3)
REAL CASH
COMMON LIST, CASH, DOLLAR, CENT
```

Если этот набор инструкций расположен в некотором программном модуле A, то во время выполнения модуля общая область памяти будет организована следующим образом:

LIST(1,1)	Слово 1
LIST(2,1)	Слово 2
LIST(1,2)	Слово 3
LIST(2,2)	Слово 4
LIST(1,3)	Слово 5
LIST(2,3)	Слово 6
CASH	Слово 7
DOLLAR	Слово 8
CENT	Слово 9

Если некоторый другой модуль B содержит такой же набор инструкций, то при каждой ссылке внутри модуля B к LIST, CASH, DOLLAR или CENT происходит обращение к тем же адресам памяти, к которым осуществляется доступ по этим именам из модуля A. Таким образом, эта область данных становится не локальной для какого-то отдельного модуля, а общей для всех модулей, которые содержат соответствующую декларацию COMMON.

На рис. 2.8 приведен простой пример использования инструкции COMMON. Программа читает данные, которые организованы следующим образом:

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* СЛИЯНИЕ ДВУХ СОРТИРОВАННЫХ СПИСКОВ. В ОЛИТОМ СПИСКЕ ДУБЛИРОВАНИЯ    *
C* НЕТ. СПИСКИ ВВОДЯТСЯ С ПЕРФОКАРТ                                    *
C*                                                                     *
C* ВВОД:                                                               *
C*   КАРТА, СОДЕРЖАЩАЯ ЧИСЛО ЭЛЕМЕНТОВ В 1-ОМ СПИСКЕ (ФОРМАТ I2)      *
C*   ПО ОДНОЙ КАРТЕ НА ЭЛЕМЕНТ 1-ГО СПИСКА (ФОРМАТ I4)                *
C*   КАРТА, СОДЕРЖАЩАЯ ЧИСЛО ЭЛЕМЕНТОВ ВО 2-ОМ СПИСКЕ (ФОРМАТ I2)    *
C*   ПО ОДНОЙ КАРТЕ НА ЭЛЕМЕНТ 2-ГО СПИСКА (ФОРМАТ I4)                *
C*                                                                     *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                           *
C*   READ, MERGE, PRINT                                                *
C*                                                                     *
C* ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                                *
C*   LIST1 - ПЕРВЫЙ СПИСОК ЦЕЛЫХ ЗНАЧЕНИЙ; РАЗМЕР 50                  *
C*   LIST2 - ВТОРОЙ СПИСОК ЦЕЛЫХ ЗНАЧЕНИЙ; РАЗМЕР 50                  *
C*   LIST3 - СПИСОК СЛИЯНИЯ LIST1 И LIST2; РАЗМЕР 100                 *
C*   S1    - ЧИСЛО ЭЛЕМЕНТОВ LIST1                                    *
C*   S2    - ЧИСЛО ЭЛЕМЕНТОВ LIST2                                    *
C*   S3    - ЧИСЛО ЭЛЕМЕНТОВ LIST3                                    *
C*                                                                     *
C* ПРОГРАММИСТ: ---                          ДАТА: ---                 *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER S1, S2, S3, LIST1(50), LIST2(50), LIST3(100)
C
C       ОБЩАЯ ОБЛАСТЬ: СПИСКИ ДОСТУПНЫ ДЛЯ MERGE И PRINT,
C       ОБЛАСТЬ НЕДОСТУПНА ДЛЯ READ
      COMMON S1, S2, S3, LIST1, LIST2, LIST3
C


C       ВВЕСТИ ДВА СПИСКА, СЛИТЬ И РАСПЕЧАТАТЬ.
C       ЕСЛИ ПРИ ВВОДЕ ОБНАРУЖЕНЫ ОШИБКИ, ОСТАНОВ
      CALL READ (&999, S1, LIST1)
      CALL READ (&999, S2, LIST2)
      CALL MERGE
      CALL PRINT
999   STOP
      END

      SUBROUTINE READ (*, SIZE, LIST)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВВЕСТИ РАЗМЕР, ЗАТЕМ ЭЛЕМЕНТЫ СПИСКА.                              *
C* ЕСЛИ РАЗМЕР НЕВЕРНЫЙ (МЕНЬШЕ 1 ИЛИ БОЛЬШЕ 50), ТО RETURN1.         *
C* ТО ЖЕ САМОЕ, ЕСЛИ КОНЕЦ ФАЙЛА                                      *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER SIZE, LIST(50), RDR, PRT, I
      DATA RDR/5/, PRT/6/
      READ (RDR,100,END=888) SIZE
100   FORMAT (I2)
      IF ((SIZE .LT. 1) .OR. (SIZE .GT. 50)) GO TO 888
      READ (RDR,101,END=888) (LIST(I), I = 1, SIZE)
101   FORMAT (I4)
      RETURN
C
C       ПРИХОДИМ СЮДА В СИТУАЦИИ ОШИБКИ
888   WRITE (PRT,889)
889   FORMAT (' *** ERROR IN DATA ***')
      RETURN1
      END
```

Рис. 2.8. Использование COMMON в программе слияния.
(Текст в инст. 889: «Ошибка в данных»; 200: «О Слитый список».)

```
      SUBROUTINE MERGE
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* СЛИЯНИЕ СПИСКОВ LIST1 И LIST2 В LIST3                                *
C*                                                                      *
C* ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                                 *
C*   LIST1 - СПИСОК ИЗ ОБЩ. ОБЛ.; ТИП ЦЕЛЫЙ; МАКС. РАЗМЕР 50          *
C*   LIST2 - СПИСОК ИЗ ОБЩ. ОБЛ.; ТИП ЦЕЛЫЙ; МАКС. РАЗМЕР 50          *
C*   LIST3 - СЛИТЫЙ СПИСОК В ОБЩ. ОБЛ.; МАКС. РАЗМЕР 100              *
C*   S1    - ФАКТИЧЕСКОЕ ЧИСЛО ЭЛЕМЕНТОВ В LIST1 (COMMON)              *
C*   S2    - ФАКТИЧЕСКОЕ ЧИСЛО ЭЛЕМЕНТОВ В LIST2 (COMMON)              *
C*   S3    - ЧИСЛО ЭЛЕМЕНТОВ В СПИСКЕ LIST3 (COMMON)                   *
C*           ВО ВРЕМЯ СЛИЯНИЯ - ЭТО PTR К ТЕКУЩЕМУ КОНЦУ LIST3        *
C*   P1    - PTR ДЛЯ LIST1, УКАЗЫВАЕТ НА ТЕКУЩИЙ ЭЛЕМЕНТ               *
C*   P2    - PTR ДЛЯ LIST2, УКАЗЫВАЕТ НА ТЕКУЩИЙ ЭЛЕМЕНТ               *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      INTEGER P1, P2, S1, S2, S3, LIST1(50), LIST2(50), LIST3(100)
      COMMON S1, S2, S3, LIST1, LIST2, LIST3
C
C       СТАРТ ОТ ВЕРШИНЫ КАЖДОГО СПИСКА
      P1 = 1
      P2 = 1
      S3 = 0
C
C       ИЗ КАКОГО СПИСКА КОПИРУЕМ ЗНАЧЕНИЕ В LIST3
10    IF (P1 .GT. S1) GO TO 30
      IF (P2 .GT. S2) GO TO 20
      IF (LIST1(P1) - LIST2(P2)) 20, 40, 35
C
C       ПРИХОДИМ СЮДА, ЕСЛИ КОПИРУЕМ ЭЛЕМЕНТ ИЗ LIST1
20        S3 = S3 + 1
          LIST3(S3) = LIST1(P1)
          P1 = P1 + 1
          GO TO 10
C
C       ОБА СПИСКА ИСЧЕРПАНЫ? ЕСЛИ ДА, ВЫХОД ИЗ ПОДПРОГРАММЫ
30    IF (P2 .GT. S2) RETURN

C       ПРИХОДИМ СЮДА, ЕСЛИ КОПИРУЕМ ЭЛЕМЕНТ ИЗ LIST2
35        S3 = S3 + 1
          LIST3(S3) = LIST2(P2)
C
C       ОПУСКАЕМ ЭЛЕМЕНТ В LIST2 (ЧТОБЫ НЕ ДУБЛИРОВАТЬ)
40        P2 = P2 + 1
          GO TO 10
      END

      SUBROUTINE PRINT
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ПЕЧАТЬ СЛИТОГО СПИСКА LIST3. РАЗМЕР СПИСКА S3                       *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER S1, S2, S3, LIST1(50), LIST2(50), LIST3(100), PRT, I, K
      COMMON S1, S2, S3, LIST1, LIST2, LIST3
      DATA PRT/6/
      WRITE (PRT,200) (LIST3(I), I = 1, S3)
200   FORMAT ('0 MERGED LIST' // (I7) )
      RETURN
      END
```

Продолжение Рис. 2.8.

Номер перфокарты	Величина	Комментарий
1	n	Целая величина в колонках 1—2
2	j_1	Целая величина в колонках 1—4
3	j_2	Как выше
.	.	. .
.	.	. .
.	.	. .
$n+1$	j_n	Как выше
$n+2$	m	Целая величина в колонках 1—2
$n+3$	k_1	Целая величина в колонках 1—4
$n+4$	k_2	Как выше
.	.	. .
.	.	. .
.	.	. .
$n+m+2$	k_i	Как выше

Затем программа производит слияние набора целых величин (j_1, . . ., j_n) и (k_1, . . ., k_i) и печатает результат слияния. (*Замечание:* в данной программе предполагается, что величины j и k отсортированы в порядке возрастания.)

Память, отводимая программе на рис. 2.8, будет организована следующим образом:

Команды главной программы	MAIN
Команды подпрограммы READ. Память для локальных переменных I, RDR, PRT. Память для констант 1 и 50. Если используется передача параметров по значению/результату, то память под SIZE.	READ
Команды подпрограммы MERGE. Память для локальных переменных P1, P2. Память для констант 1 и 0.	MERGE
Команды подпрограммы PRINT. Память для локальных переменных I, PRT. Память для константы 1.	PRINT

S1	Слово 1	COMMON
S2	Слово 2	
S3	Слово 3	
LIST1	Слова с 4 по 53	
LIST2	Слова с 54 по 103	
LIST3	Слова с 104 по 203	

2.6.3. Несогласующиеся определения COMMON

Во многих языках программирования переменные и массивы считаются глобальными, если доступ к одной и той же ячейке памяти осуществляется из разных модулей по одному и тому же имени. В Фортране, однако, глобальные переменные и массивы ассоциированы с адресами памяти не по именам (имена могут быть любыми), а по месту в общем блоке. Рассмотрим вновь модули A и B из предыдущего раздела, полагая, что модуль C содержит следующие инструкции:

```
INTEGER UNIT1, UNIT2, ARRAY(2, 3)
REAL MONEY
COMMON ARRAY, MONEY, UNIT1, UNIT2
```

В этом случае общий блок для модуля C будет организован так:

ARRAY(1,1)	Слово 1
ARRAY(2,1)	Слово 2
ARRAY(1,2)	Слово 3
ARRAY(2,2)	Слово 4
ARRAY(1,3)	Слово 5
ARRAY(2,3)	Слово 6
MONEY	Слово 7
UNIT1	Слово 8
UNIT2	Слово 9

но не так:

LIST(1,1)	Слово 1
LIST(2,1)	Слово 2
LIST(1,2)	Слово 3
LIST(2,2)	Слово 4
LIST(1,3)	Слово 5
LIST(2,3)	Слово 6
CASH	Слово 7
DOLLAR	Слово 8
CENT	Слово 9

Последнее представляет собой распределение памяти для модулей A и B, а не для модуля C. Теперь значения массива ARRAY модуля C будут совпадать со значениями массива LIST модулей A и B, поскольку эти массивы ассоциируются с одной и той же областью памяти. Ссылки на MONEY, UNIT1 и UNIT2 соответствуют ссылкам на CASH, DOLLAR и CENT.

Рассмотрим еще более крайний случай. Пусть модуль D содержит инструкции

```
INTEGER L, Q
REAL A(2), S(3)
COMMON L, A, Q, S
```

которые задают следующее распределение общей области:

L	Слово 1
A(1)	Слово 2
A(2)	Слово 3
Q	Слово 4
S(1)	Слово 5
S(2)	Слово 6
S(3)	Слово 7

Ссылка в модуле D на переменную L обеспечит в результате доступ к тому же слову памяти, что и ссылка на LIST(1, 1) в модуле A и B или на ARRAY(1, 1) в модуле C; для A(1) отводится та же ячейка памяти, что и для LIST(2, 1) и ARRAY(2, 1), и т. д. Ссылки на DOLLAR и CENT в модулях A и B, UNIT1 и UNIT2 в модуле C не имеют аналога в модуле D, поскольку этот модуль связан только с первыми семью словами блока COMMON.

Ясно, что несоответствия в описании общей области делают программу трудной для понимания. Более того, такая несогласованность часто приводит к ошибкам. Чтобы избежать потенциальных затруднений и сделать программу более удобной для чтения, мы рекомендуем соблюдать следующее правило:

Все модули, которые содержат инструкции COMMON, должны иметь согласованные описания структуры общей области.

Лучший способ для достижения этого — подготовить один «экземпляр» общих деклараций, включая соответствующие декларации типов REAL, INTEGER и LOGICAL, а затем размножить этот экземпляр для каждого модуля, в котором используется блок COMMON.

2.6.4. Блок COMMON с именем

Рассмотренный нами в предыдущих двух подразделах тип COMMON называется *безымянным* или *непомеченным* общим блоком. Можно определить также другие, независимые общие блоки, называемые *именными* или *помеченными* общими блоками. Набор переменных и массивов, относящихся к общей области с именем, указывается в инструкции вида

COMMON *(имя) список переменных и массивов*

Перед выполнением программы отводится по одному блоку памяти для каждой отдельной помеченной общей области, которая была объявлена внутри какого-либо из программных модулей.

Блок COMMON с именем часто используют в программах, состоящих из двух или более наборов подпрограмм, таких, что

подпрограммы одного набора взаимодействуют друг с другом и не взаимодействуют с подпрограммами других наборов. Каждый набор в этом случае имеет свою собственную помеченную общую область. При использовании этого способа (вместо слияния всех общих областей в один блок) снижается вероятность непреднамеренного изменения какими-либо подпрограммами общих данных. Ошибки в этом случае с большей вероятностью будут носить локальный характер.

2.6.5. Взаимосвязь инструкций COMMON и EQUIVALENCE

Переменные или массивы внутри общей области могут быть объявлены эквивалентными каким-либо локальным переменным или массивам. Если соответствующая эквивалентность установлена, то память для такой локальной переменной или массива не выделяется, поскольку она уже отведена в общем блоке. Эквивалентности такого вида могут быть использованы во многих ситуациях, рассмотренных в гл. 1 (например, при обработке многомерных массивов как ряда одномерных с целью повышения эффективности). Существует, однако, одно ограничение при установлении подобной эквивалентности: эквивалентная локальная область не должна «сдвигать влево» начало общего блока. Поясним на примере следующей последовательности инструкций:

```
INTEGER A(20), B(16), C, D, E(6)
COMMON B, C, D
EQUIVALENCE (C,E(1)), (C,A(20))
```

Общая область включает массив B, переменные C и D и имеет длину 18 слов. Она начинается с элемента B(1). Первая пара в декларации EQUIVALENCE устанавливает, что переменная C и элемент массива E(1) ассоциируются с одним и тем же словом памяти. Так как D в общей области непосредственно следует за C, то возникает соответствие между D и E(2). Хотя это плохой способ установления соответствий, но такая эквивалентность допустима. Возникает побочный эффект, заключающийся в расширении общей области от 18 до 22 слов, т. е. изменяется «правая граница» общей области.

Вторая пара в декларации EQUIVALENCE (C, A(20)) задана неверно. Так как элемент B(16) в общей области непосредственно предшествует переменной C, то B(1) становится эквивалентным A(4). Это приводит к тому, что элементы A(1), A(2) и A(3) должны предшествовать элементу B(1), в связи с чем

происходит сдвиг начала общего блока. Так как общему блоку может предшествовать в памяти другой общий блок или область, принадлежащая к какому-либо программному модулю, то это может привести к непредсказуемым последствиям и даже к уничтожению команд программы.

Можно установить эквивалентность между переменными общих областей и локальными переменными, но не разрешается устанавливать эквивалентность общей области с формальными параметрами или другой общей областью. Первое из этих ограничений логично, так как формальные параметры должны быть ассоциированы с соответствующими им фактическими параметрами, а не с какими-либо фиксированными общими переменными или массивами. В целесообразности второго ограничения можно убедиться на следующем примере:

```
INTEGER A, B, C, D, E
COMMON/COM1/ A, B
COMMON/COM2/ C, D
EQUIVALENCE (A,B), (A,C), (D,E), (B,E)
```

Первая пара (A, B) содержит противоречие. Внутри общего блока COM1 две переменные нельзя описывать как эквивалентные, поскольку они связаны с разными словами этого блока. Переменной B предписано находиться одновременно в двух участках памяти. Вторая пара (A, C) также нелогична, так как общие блоки COM1 и COM2 представляют собой раздельные, неперекрывающиеся области памяти. Хотя третья и четвертая пары по отдельности правильны, в совокупности они не верны, так как их объединение в инструкции EQUIVALENCE приводит к тому, что переменным B и D отводится одно и то же слово памяти.

Просуммируем теперь ограничения, действующие при использовании EQUIVALENCE с элементами данных, принадлежащими общим блокам COMMON:

1. Эквивалентность не должна «сдвигать влево» начало общего блока.
2. Переменные или массивы одного общего блока нельзя описывать как эквивалентные.
3. Переменные или массивы в общем блоке не могут быть эквивалентны формальным параметрам.

2.6.6. Инициализация элементов данных в общих блоках

По правилам Фортрана не разрешается инициализация любых переменных или элементов массивов, входящих в общий блок без имени. Однако данные можно инициализировать в

помеченном общем блоке. Чтобы это сделать, следует создать неисполняемый программный модуль, называемый подпрограммой BLOCK DATA.

Подпрограмма BLOCK DATA начинается с инструкции

```
BLOCK DATA
```

завершается инструкцией END и не содержит ничего, кроме COMMON с именем, деклараций типа, DIMENSION и DATA. Приведем пример подпрограммы BLOCK DATA:

```
BLOCK DATA
INTEGER A, B, C(3), D(2,2)
REAL E(4,2), Q, R
COMMON/PASS1/ A, B, E
COMMON/PASS2/ C, D, Q, R
DATA A/15/, E/8*0.0/, D/1,2,3,4/
END
```

Здесь указано, что помеченный общий блок PASS1 содержит три элемента A, B и E. A и E инициализируются начальными значениями, а B — нет. Общий блок PASS2 состоит из четырех элементов C, D, Q и R. Только переменной D присваивается начальное значение, в то время как переменные C, Q и R имеют непредсказуемые значения.

2.7. Функция-оператор

Помимо процедур-функций, язык Фортран позволяет программисту определять простые функции, внутренние по отношению к программному модулю, в котором они определены и в котором они вызываются. Эти функции называют *функциями-формулами* или *функциями-операторами*. Каждая функция-оператор определяется одиночной инструкцией присваивания и помещается после всех деклараций перед первой исполняемой инструкцией программного модуля. Общая форма такой инструкции:

$$\textit{имя}\ (a_1,\ a_2,\ \ldots,\ a_n)=\textit{выражение}$$

где *имя* — имя какой-либо переменной, a_1, a_2, . . ., a_n — формальные параметры и *выражение* — любое арифметическое или логическое выражение. *Выражение* не должно содержать ссылок на элементы массивов или на функцию-оператор, которая ранее не была определена внутри того же модуля. *Имя* определяет тип функции.

Функция-оператор вызывается подобно любым другим функциям. Встретившись с вызовом такой функции, компилятор действует таким же образом, как если бы мы просто подставили соответствующее выражение, заменив каждый формальный па-

раметр фактическим параметром. Например, если мы определяем функцию SAMPLE как

```
SAMPLE (Y, Z)=Y*(Z+Y)**2—L
```

то инструкция

```
RESULT=A+SAMPLE(A, B)
```

компилируется так же, как если бы мы записали

```
RESULT=A+(A*(B+A)**2—L)
```

Заметим, что переменная L никак не связана с фактическими параметрами, так как она не входит в число формальных параметров.

Функции-операторы не уменьшают потребности в памяти, что обычно достигается при использовании процедур функций. Их основное достоинство состоит в том, что программы, в которых содержатся функции-операторы, легче читать, они позволяют сократить утомительное написание общеупотребимых арифметических и логических выражений. В свою очередь это приводит к уменьшению количества ошибок, так как программисты обычно становятся менее бдительными, если работа носит монотонный характер. Применение функций-операторов может помочь при изменении программы, например, если понадобится изменить что-либо в часто встречающихся вычислениях.

В качестве примера использования функции-оператора рассмотрим способ эффективного хранения в памяти матрицы A, в которой все значения выше главной диагонали равны нулю (т. е. если J>I, то A(I, J)=0). Такая матрица называется *нижней треугольной матрицей* и используется довольно часто. Если A — нижняя треугольная матрица размером 50 на 50, то распределение памяти посредством инструкции

```
DIMENSION A(50, 50)
```

расточительно, так как из 2500 слов 1225 будут содержать нулевые значения. Более эффективным оказывается распределение памяти при помощи инструкции

```
DIMENSION A(1275)
```

В этом случае на (I, J)-компоненту A(I⩾J) можно ссылаться как на A((I*(I—1))/2+J). Определим функцию-оператор

```
INDEX (I, J)=(I*(I—1))/2+J
```

Тогда при ссылке на компоненту (I, J) мы будем писать A(INDEX (I, J)), если, конечно, I⩾J и если данный диалект языка позволяет вызывать функции в индексном выражении.

2.8. Заключительный пример — генераторы случайных чисел

2.8.1. Понятие псевдослучайной последовательности

Когда приходится отлаживать сложную программу, одна из наиболее трудных задач — сгенерировать подходящие пробные значения данных. Пусть, например, нужно сгенерировать номера социального страхования, которые необходимо ввести в тестируемую программу. Мы, конечно, не будем вводить все возможные числа, так как общее их количество составляет 1 миллиард. Вместо этого постараемся подобрать какое-то ограниченное количество пробных значений (например, 1000 чисел). Хотелось бы, чтобы эти значения были случайным образом распределены по всему диапазону возможных чисел. Если бы все пробные номера социального страхования находились в диапазоне от 0 по 999, это не дало бы удовлетворительных результатов.

Один из методов, посредством которого мы могли бы решить нашу задачу: бросать 10-гранную игральную кость с гранями, пронумерованными от 0 до 9. Каждый бросок игральной кости дает случайное число. Девять бросков дают нам случайно выбранный номер социального страхования, а 9000 бросков обеспечивают 1000 необходимых тестовых чисел. Вопрос состоит в том, как написать программу, которая могла бы генерировать случайные числа, ведь все машинные программы выполняются в детерминированной (неслучайной) форме. Для того чтобы понять, как можно написать такую программу, вначале рассмотрим, какими свойствами должна обладать случайная последовательность.

Вообще говоря, события можно считать случайными, когда каждое событие (выборка числа) не зависит от каких бы то ни было предшествующих событий. В этом смысле даже такая последовательность из 1000 чисел, каждый член которой равен 555026281, может быть результатом случайной генерации 9-разрядных десятичных чисел. Когда мы используем термин «случайный», по существу мы имеем в виду однородное распределение последовательности чисел в требуемом диапазоне (от 0 по 999999999). Мы, вероятно, ожидали бы, что из 1000 чисел примерно 500 находились бы в диапазоне от 0 до 499999999. Более того, мы не ожидали бы часто повторяющихся сочетаний, периодичности и т. д. Например, последовательность 0, 5, 10, 15, . . . не приемлема, хотя ее числа равномерно распределены. Последовательности, удовлетворяющие вышеописанным критериям, часто называют *псевдослучайными*, так как они внешне выглядят случайными, но не содержат аномалий, которые могут иметь место в действительно случайных последовательностях.

2.8.2. Мультипликативный генератор случайных чисел

В области методов генерации случайных последовательностей выполнено множество исследований [1]). К числу изученных и найденных вполне удовлетворительными методов относятся так называемые *мультипликативные* методы.

Генерируемая последовательность определяется стартовым числом $r(0)$, называемым *началом отсчета*, и формулой

$$r(k+1)=(a*r(k)) \qquad \text{по модулю } (m),$$

вычисляющей следующее число последовательности. Так, первое число такой последовательности равно

$$r(1)=(a*r(0)) \qquad \text{по модулю } (m),$$

второе число

$$r(2)=(a*r(1)) \qquad \text{по модулю } (m)$$

и т. д. Свойства последовательности, генерируемой этим методом, существенно зависят от выбора значений $r(0)$, a и m. Хороший выбор этих значений для одной ЭВМ может оказаться совершенно неудовлетворительным для других ЭВМ. Часто значение m выбирают как значение, превышающее наибольшую целую величину, представимую на данной ЭВМ. Такой выбор имеет два достоинства: во-первых, он позволяет распределять числа в максимально большом возможном диапазоне и, во-вторых, модуль функции для такого значения m может быть тривиально вычислен на различных ЭВМ.

2.8.3. Интегрирование по методу Монте-Карло

Теперь мы приведем программу, в которой случайные числа используются для приближенного вычисления площади под некоторой кривой (см. рис. 2.6, где был продемонстрирован другой метод решения этой задачи). Метод, к которому мы обратимся здесь, обычно называют методом *Монте-Карло*. Он позволяет аппроксимировать площадь под кривой f в интервале $[a, b]$. Нарисуем прямоугольник с вершинами $(a, 0)$, (a, h), (b, h) и $(b, 0)$. Площадь такого прямоугольника $S_{\text{прямоуг}}=h*(b-a)$. Площадь под кривой f в интервале $[a, b]$ можно аппроксимировать произведением $S_{\text{прямоуг}}*P_f$, где P_f — приближенно вычисленная часть прямоугольника, расположенная под кривой f (рис. 2.9). Метод интегрирования Монте-Карло, представленный

---

[1]) См., например, Кнут Д. Искусство программирования для ЭВМ, т. 2, гл. 3 — М.: Мир, 1977.— *Прим. ред.*

на рис. 2.10, позволяет оценить значение P_f путем генерации большого числа случайных точек (1000) внутри прямоугольника и вычисления процентной доли тех из них, которые попали в область ниже кривой f.

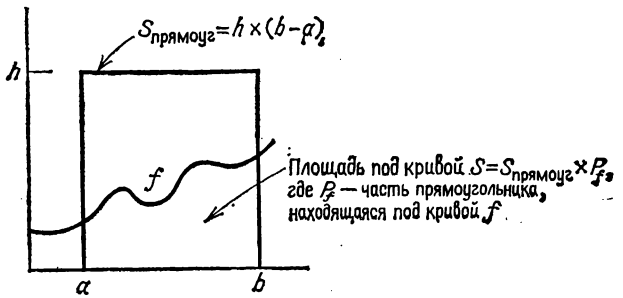


Рис. 2.9. Интегрирование по методу Монте-Карло.

Генератор случайных чисел RAND, приведенный в данном примере, мультипликативный. Он предназначен для использования на машинах IBM 360-370. Генератор вырабатывает вещественные числа в интервале [0, RMAX], причем RMAX служит в качестве его единственного аргумента. Вначале генератор вырабатывает случайное целое число g в диапазоне от 1 до $2^{31}-1$ (число $2^{31}-1$ является наибольшим целым числом, представимым на IBM 360-370). Число g затем нормализуется в интервале от 0

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* АППРОКСИМАЦИЯ ПЛОЩАДИ ПОД КРИВОЙ X**2-X В ИНТЕРВАЛЕ ОТ 1 ДО 4.      *
C* ИСПОЛЬЗУЕТСЯ МЕТОД МОНТЕ-КАРЛО.                                      *
C*                                                                      *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ: AREA                                        *
C*                                                                      *
C* ПРОГРАММИСТ: ---                                ДАТА: ---            *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER PRT
      REAL APPROX, AREA
      DATA PRT/6/
C
C       ФУНКЦИЯ ОПРЕДЕЛЯЕТСЯ ПОДПРОГРАММОЙ FCN
C       ИМЯ FCN ПЕРЕДАЕТСЯ В КАЧЕСТВЕ ПАРАМЕТРА
      EXTERNAL FCN
C
C       ВЫЧИСЛИТЬ ПЛОЩАДЬ И НАПЕЧАТАТЬ РЕЗУЛЬТАТ.
C       ВЕЛИЧИНА 12, ПЕРЕДАВАЕМАЯ В КАЧЕСТВЕ 1-ГО ПАРАМЕТРА, УКАЗЫВАЕТ
C       НАИБОЛЬШЕЕ ЗНАЧЕНИЕ В ИНТЕРВАЛЕ ОТ 1 ДО 4.
C       ЭТО НУЖНО ДЛЯ АЛГОРИТМА ИНТЕГРИРОВАНИЯ
      APPROX = AREA(12., 1000, 1., 4., FCN)
      WRITE (PRT,200) APPROX
200   FORMAT ('0', 5X, 'THE AREA UNDER X**2-X FROM 1 TO 4 IS', F10.2)
      STOP
      END
```

Рис. 2.10. Программа интегрирования по методу Монте-Карло.
(Текст в инстр. 200: «Площадь под кривой X**2—X от 1 до 4».)

```
      REAL FUNCTION FCN (X)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВОЗВРАЩАЕТ ЗНАЧЕНИЕ X**2-X                                        *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      REAL X
      FCN = X**2 - X
      RETURN
      END
      REAL FUNCTION AREA (HIGH, A, B, F)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИНТЕГРИРУЕТ ПО МЕТОДУ МОНТЕ-КАРЛО ПЛОЩАДЬ ПОД ФУНКЦИЕЙ 'F'       *
C* В ИНТЕРВАЛЕ ОТ A ДО B.                                             *
C* ПАРАМЕТРЫ:                                                         *
C*    HIGH  - НАИБОЛЬШЕЕ ЗНАЧЕНИЕ F В ИНТЕРВАЛЕ ОТ A ДО B             *
C*            (ТИП REAL)                                              *
C*    N     - ЧИСЛО СЛУЧАЙНЫХ ТОЧЕК НА ПЛОСКОСТИ ДЛЯ ВЫЧИСЛЕНИЯ      *
C*            ПЛОЩАДИ ПОД КРИВОЙ F (INTEGER)                         *
C*    A     - НАЧАЛО ИНТЕРВАЛА (ТИП REAL)                             *
C*    B     - КОНЕЦ ИНТЕРВАЛА (REAL)                                  *
C*    F     - ФУНКЦИЯ, ПОД КОТОРОЙ ВЫЧИСЛЯЕТСЯ ПЛОЩАДЬ (REAL)        *
C*                                                                    *
C* ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                               *
C*   HITS  - ЧИСЛО СЛУЧАЙНЫХ ТОЧЕК ПОД КРИВОЙ (INTEGER)               *
C*   INTVAL - РАССТОЯНИЕ МЕЖДУ B И A (REAL)                           *
C*                                                                    *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                           *
C*    RAND, FLOAT                                                     *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER N, I, HITS
      REAL HIGH, A, B, INTVAL, X, Y
C
C       ВЫЧИСЛИТЬ N СЛУЧАЙНЫХ ТОЧЕК И ПОДСЧИТАТЬ
C       ЧИСЛО „ПОПАДАНИЙ" (ТОЧКИ ПОД КРИВОЙ F)
      INTVAL = B - A
      HITS = 0
      DO 10 I = 1, N
         Y = RAND(HIGH)
         X = RAND(INTVAL) + A
         IF (F(X) .GE. Y) HITS = HITS+1
10    CONTINUE
C       ПЛОЩАДЬ ВЫЧИСЛЯЕТСЯ КАК ОТНОСИТЕЛЬНОЕ ЧИСЛО „ПОПАДАНИЙ",
C       УМНОЖЕННОЕ НА ПЛОЩАДЬ ПРЯМОУГОЛЬНИКА, ОГРАНИЧЕННОГО ТОЧКАМИ
C       (A,0),(A,HIGH),(B,HIGH),(B,0)
      AREA = HIGH*INTVAL*(FLOAT(HITS)/N)
      RETURN
      END

      REAL FUNCTION RAND (RMAX)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВЫЧИСЛЯЕТ СЛУЧАЙНОЕ ВЕЩЕСТВЕННОЕ ЧИСЛО В ИНТЕРВАЛЕ ОТ 0 ДО RMAX.  *
C* ПРОГРАММА ГОДИТСЯ ДЛЯ 32-РАЗРЯДНОГО ДОПОЛНИТЕЛЬНОГО КОДА.         *
C* В ЧАСТНОСТИ, ОНА РАБОТАЕТ НА IBM 360-370                           *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      REAL RMAX
      INTEGER IRAND, MULT, LARGE
      DATA IRAND/137462873/, MULT/65539/, LARGE/2147483647/
C
C       ВЫЧИСЛИТЬ НОВОЕ СЛУЧАЙНОЕ ЧИСЛО (ОТ 1 ДО 2**31-1)
      IRAND = IRAND * MULT
      IF (IRAND .LT. 0) IRAND = (IRAND + LARGE) + 1
C
C       ВЫЧИСЛИТЬ ТРЕБУЕМОЕ ВЕЩЕСТВЕННОЕ ЧИСЛО
      RAND = RMAX*(IRAND-1) / (LARGE-1)
      RETURN
      END
```

Продолжение Рис. 2.10.

до RMAX. Нормализация выполняется вычислением значения $((g-1)/(2^{31}-2))*RMAX$.

При изучении приведенного примера следует помнить, что генератор случайных чисел RAND предназначен специально для машин IBM 360-370. Он может не подойти для другой ЭВМ. Следует также знать, что, хотя методы Монте-Карло могут давать очень хорошие результаты, их сходимость обычно намного медленнее, чем сходимость других методов, таких, например, как метод трапеций.

Упражнения

2.01. Проанализируйте ситуации, которые возникают при изменении первого параметра следующей функции *M*:

```
      J = 2
      I = 0
      IF ((I.EQ.0) .OR. (M(I,J).EQ.J)) WRITE (6,20) I
20    FORMAT (I12)
      STOP
      END

      INTEGER FUNCTION M(I,J)
      IF (J.GT.I) I=J
      M = I
      RETURN
      END
```

2.02. Рассмотрите следующую программу:

```
      COMMON I,J
      L = 2
      J = 3
      CALL SUB (I,L)
      WRITE (6,20) I,J,L
20    FORMAT (3I12)
      STOP
      END

      SUBROUTINE SUB (K,L)
      COMMON I,J
      I = L
      K = J + 1
      L = I
      RETURN
      END
```

Какие значения будут напечатаны для переменных I, J и L, если

а) I и L передаются в подпрограмму SUB *по значению / результату*?

б) I и L передаются в подпрограмму SUB *по адресу* (*ссылке*)?

2.03. Используя альтернативные выходы, напишите инструкцию CALL и подпрограмму, которые выполняют следующие действия. Инструкция CALL передает числовые параметры 3.*X**2, 4.*Z, 2.*(X—Z) и ANS. Подпрограмма имеет соответствующие формальные параметры A, B, C и D. Она вычисляет значение D=B**2—4*A*C. Если значение D положительно, выполняется

нормальный возврат. Если D равно нулю, подпрограмма передает управление на метку 20 вызывающей программы. Если D отрицательно, управление передается инструкции с номером 99.

2.04. Напишите вещественную функцию, имеющую две точки входа SUM и ABSSUM и два параметра A и N. Если происходит вход по имени SUM, то возвращается сумма первых N элементов вещественного списка A. При входе через ABSSUM возвращается абсолютное значение этой суммы.. Подпрограмма, которую вы напишете, должна содержать только один цикл, суммирующий элементы A. Проверьте работу вашей подпрограммы с соответствующей главной программой.

2.05. Напишите вещественную функцию с двумя точками входа, называемыми FMAX и FMIN, и четырьмя параметрами F, A, B и N, где F — вещественная функция, A и B являются вещественными значениями, представляющими начало и конец некоторого интервала, N — целая переменная. При входе через точку FMAX должно возвращаться приближенное максимальное значение, которого достигает F в интервале от A до B. Функция FMIN используется для вычисления минимума. Аппроксимации FMAX и FMIN вычисляются путем анализа значений F в точках A, A+(B—A)/N, . . ., B—(B—A)/N, B. Проверьте работу функции с помощью соответствующей главной программы и нескольких значений вещественных параметров функции.

2.06. Программа на рис. 2.6 вычисляет по методу трапеций площадь под некоторой кривой F на интервале (A, B). *Метод средней точки* подобен методу трапеций. По этому методу интервал (A, B) делится на N подынтервалов. Однако здесь площадь в подынтервале (C, D) аппроксимируется по формуле (D—C)*F ((C+D)/2). Напишите программу, которая сравнивает эти методы, для функций X**3—X**2 и X*SQRT (X) в интервале от 2 до 10. В каждом случае делите этот интервал на 24 подынтервала.

2.07. Если машина, на которой вы работаете, отлична от IBM 360-370, посмотрите, годится ли для вашей машины генератор случайных чисел, приведенный на рис. 2.10. Если нет, напишите такую программу сами.
Замечание. Во многих вычислительных центрах эта программа входит в библиотеку Фортрана.

2.08. Напишите и проверьте подпрограмму RNDSEQ, которая имеет два целых параметра LIST и N, где LIST — это массив размера N.RNDSEQ используется в качестве генератора случайных чисел и вырабатывает последовательность N случайных целых чисел $r(1)$, $r(2)$, . . ., $r(N)$. Каждое число должно находиться в диапазоне от 1 до N и не должно повторяться. При возврате данная подпрограмма должна присваивать значение $r(1)$ элементу LIST(1), значение $r(2)$ элементу LIST(2) и т. д. *Предупреждение.* Вследствие того, что какие-то числа могут повторяться, в данном примере может понадобиться более N обращений к генератору случайных чисел.

2.09. Метод хеш-таблиц, рассмотренный в разд. 2.4, разрешает коллизии посредством техники линейного опробирования. Это не самый лучший метод. Метод *случайного опробирования* генерирует последовательность $n-1$ одиночных случайных чисел $r(1)$, $r(2)$, . . ., $r(n-1)$ в диапазоне от 1 до $n-1$. При возникновении коллизии в позиции $h(k)$ мы пытаемся воспользоваться $h(k)+r(1)+h(k)+r(2)$, . . ., $h(k)+r(n-1)$. Как и при линейном опробировании, хеш-таблица рассматривается как циклическая. Напишите программу, которая сравнивает число проб, необходимых при линейном и случайном опробировании. Создайте несколько наборов данных для выполнения этого сравнения. *Предложение.* Генератор случайных чисел является идеальным средством для создания таких тестовых данных.

2.10. Предположим, что некоторый программный модуль содержит инструкции

```
INTEGER LIST(3), MATRIX(2,3)
REAL COEFF, EIGEN(2)
COMMON LIST, COEFF, MATRIX, EIGEN
```

Какие переменные и элементы массивов ассоциируются с каждым из 12 слов общей области?

2.11. Рассмотрите приведенный ниже набор инструкций. Укажите, какие инструкции правильны, а какие ошибочны. Объясните, почему они ошибочны. В правильных инструкциях прокомментируйте возможные побочные эффекты.

a)
```
INTEGER J(5), K, L(13,2)
REAL M(30)
COMMON J, K, L
EQUIVALENCE (L(1,1),M(1))
```

b)
```
INTEGER J(5), K, L(13,2), M(30)
COMMON J, K, L
EQUIVALENCE (K,M(7))
```

c)
```
INTEGER J(5), K, L(13,2), M(30)
COMMON J, K, L
EQUIVALENCE (J(1),M(1)), (L(1,1),M(8))
```

d)
```
INTEGER LIST(3), MATRIX(2,3), SIX(6)
REAL COEFF, EIGEN(2), FIVE(5)
COMMON LIST, COEFF, MATRIX, EIGEN
EQUIVALENCE (SIX(1),MATRIX(2,2))
```

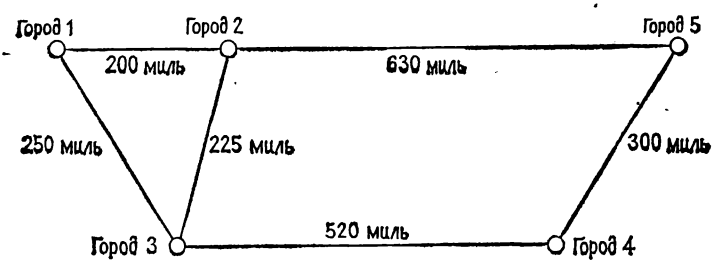
e)
```
INTEGER LIST(3), MATRIX(2,3), SIX(6)
REAL COEFF, EIGEN(2), FIVE(5)
COMMON LIST, COEFF, MATRIX, EIGEN
EQUIVALENCE (SIX(2),FIVE(3))
EQUIVALENCE (FIVE(1),LIST(1))
EQUIVALENCE (SIX(1),MATRIX(1,1))
```

2.12. Каково содержимое массива A размером 2×2, который распечатывается с помощью программы

```
      INTEGER A(2,2), I, J, ARG, Z, FN, FILL
      COMMON Z
      FILL (ARG) = FN(3-ARG)
      Z = 0
      DO 2 I = 1,2
         DO 1 J = 1,2
            A(I,J) = FILL(I+J)
1        CONTINUE
2     CONTINUE
      WRITE (6,20) A
20    FORMAT (1X,2I7)
      STOP
      END
```

```
INTEGER FUNCTION FN(P)
INTEGER P, Z
COMMON Z
FN = P
IF (Z .LE. P) RETURN
FN = Z
Z = Z+1
RETURN
END
```

2.13. Можно представить план расстояний между городами с помощью массива целых величин. Рассмотрим в качестве примера следующий план:



Этот план может быть представлен двумерным массивом (5×5)

0	200	250	-1	-1
200	0	225	-1	630
250	225	0	520	-1
-1	-1	520	0	300
-1	630	-1	300	0

Здесь —1 обозначает неизвестное расстояние. Программа считывает вначале целое число N ($N \leqslant 5$), а затем матрицу PATHS целых чисел размером N×N. Далее ваша программа проверяет, является ли PATHS правильным представлением плана (т. е. проверяет, чтобы PATHS (I, J)=PATHS(J,I) для всех значений I, $J \leqslant N$). Если нет, обработка оканчивается. В противном случае все —1 должны быть заменены на кратчайшие расстояния между соответствующими двумя городами. Следует быть осторожным, так как такого пути может и не оказаться. В этом случае нужно оставить значение —1. В завершение массив PATHS должен быть распечатан.

2.14. Напишите программу управления учетом товаров (инвентаризации). Основанная на заранее определенных входных командах программа должна обновлять инвентарный файл при новых поступлениях и продаже товаров, печатать инвентарный отчет и запросы на товары. Входные данные в программу начинаются с набора инвентарных карт, каждая из которых имеет следующую форму:

Колонка	*Вид данных*
1—9	Номенклатура товара
11—14	Количество в наличии
16—19	Проданное количество
21—27	Оптовая цена
31—37	Розничная цена

Конец инвентарных карт отмечается картой, которая содержит отрицательное число в поле номенклатуры. После ввода инвентарных карт следует набор управляющих карт, каждая из которых имеет такую структуру:

Колонка	*Вид данных*
1	Код протокола
6—14	Номенклатура товара
16—19	Количество
21—27	Оптовая цена
31—37	Розничная цена

Здесь код протокола интерпретируется следующим образом.

Код-1 — Произведена закупка. Файл надо обновить. Если данная номенклатура ранее не появлялась, то нужно добавить ее к списку, отсортированному по номерам. Если данная номенклатура уже имеется в списке, можно игнорировать розничную и оптовую цены, поскольку они должны оставаться постоянными, т. е. теперь для обновления файлов требуется лишь поле количества товара Распечатайте суммирующее сообщение по закупкам.

Код-2 — Должна быть произведена продажа. Колонка 21 и все последующие игнорируются. (Эти колонки должны быть заполнены пробелами.) Проверьте, хватит ли для этой продажи товара. Если нет, то предположите, что будет продан весь оставшийся запас этого товара (если он имеется). В любом случае нужно соответственно уменьшить имеющееся количество данного товара и увеличить общее количество проданных товаров.

Код-3 — Следует распечатать список всех товаров, которые должны быть заказаны, потому что их мало в наличии. Критерий: новая партия товара заказывается, если имеющийся запас меньше или равен трем единицам. Пример сообщения о необходимых закупках задан форматом FORMAT1.

Код-4 — Следует напечатать полный инвентарный отчет. Пример формы инвентарного отчета показан в формате FORMAT 2.

Код-5 — Остановить обработку. *Замечание.* В реальной системе инвентаризации следует предусмотреть вывод текущего инвентарного файла для его дальнейшего использования,

FORMAT 1

Запрос на покупку товара

Номенклатура	Имеющееся количество товара	Общее кол-во проданного товара	Оптовая цена	Розничная цена
57256	3	10	28.75	38.88
64031	0	623	14.25	26.99

FORMAT 2

Инвентарный отчет

Номенклатура	Имеющееся количество товара	Общее кол-во проданных товаров	Оптовая цена	Розничная цена
57256	3	10	28.75	38.99
57921	542	67	4.75	6.95
64031	0	623	14.25	26.99
96021	4	0	425.00	625,00

Глава 3

ПРОГРАММИРОВАНИЕ НЕЧИСЛЕННЫХ ЗАДАЧ

3.1. Литеры в качестве данных

Язык Фортран был разработан в основном для решения так называемых научных задач. По этой причине в языке предусмотрена операция возведения в степень; подпрограммы извлечения квадратного корня и вычисления тригонометрических функций поставляются вместе с любым компилятором Фортрана. Наряду с этим обработка текстовой информации представляет собой одну из важнейших областей применения ЭВМ. Фортран не имеет столь мощных средств обработки текстовой информации, какие он имеет для вычислений. В этой главе мы объясним основы обработки текстов, приведем ряд практических методов, которыми пользуются программисты в решении задач, связанных с обработкой текстов. Вы можете пользоваться этими средствами и разрабатывать свои собственные.

Прежде всего несколько слов о терминологии. Будем называть любой отдельный символ *литерой*. Литерой считается любая буква (от A до Z), цифра (от 0 до 9) или любой из знаков, таких, как (),.= и пробел. (Будем представлять пробел буквой b в тех случаях, когда этот символ необходимо явно указать.) Знаки часто называют *специальными литерами*. Устройства ввода и вывода, с которыми вы, возможно, знакомы, распознают все 26 прописных букв латинского алфавита, 10 цифр и некоторое число специальных литер. Ряд сложных устройств ввода/вывода позволяет использовать строчные буквы, цифровые индексы, буквы греческого алфавита и другие специальные литеры для различных применений. Последовательность нескольких литер называют *цепочкой литер* или просто *цепочкой*. *Длина* цепочки — это число литер, которые ее составляют.

3.2. Кодирование литер

Необходимо понимать, что вся информация, хранящаяся в памяти машины, представлена в виде цепочек битов. При интерпретации и использовании эти цепочки битов наделяются конкретным смыслом. Существует несколько способов кодирования информации в ЭВМ, с помощью которых литера записывается в виде последовательности нулей и единиц. Например,

цепочка из четырех литер '1020' кодируется в одном слове машины IBM 360-370 так:

11110001	11110000	11110010	11110000	цепочки битов
'1'	'0'	'2'	'0'	литеры

Заметим, что нет явного соответствия между представлением цепочки цифровых литер и эквивалентного ей целого или вещественного числа. (Десятичное целое число 1020 было бы представлено на той же самой машине в двоичном коде как 00000000 00000000 00000011 11111100.)

Обычно применяют три способа кодирования литер: BCD (двоично-десятичный код), при котором каждая из 64 различных литер представлена 6-разрядным двоичным кодом; ASCII (Американский стандартный код для передачи информации), при котором каждая из 128 различных литер выражена 7-разрядным двоичным кодом; EBCDIC (расширенный двоично-кодированный десятичный код для передачи информации), использующий 8 двоичных разрядов (битов) для представления каждой из 256 различных литер. Один из этих кодов, возможно, применен в той реализации Фортрана, с которой вы работаете. Однако программисту, работающему на Фортране, обычно не нужно знать, какой метод кодирования принят в машине. Можно всегда написать программу, которая будет правильно работать при всех трех способах кодирования литер.

Возьмем два слова, содержащих литеры, и сравним их между собой так, как если бы они содержали арифметические (например, целые величины) значения. Таким путем мы установим упорядочение литер. При всех трех способах кодирования литер целое число, представляющее литеру A, будет меньше, чем число, представляющее литеру B, которое в свою очередь меньше числа, представляющего литеру Z. Код литеры 0, интерпретируемый как целое число, дает значение, меньшее 1, которое в свою очередь меньше, чем значение кода литеры 9. Упорядочение литер от наименьшей к наибольшей при рассмотрении их как целых чисел, называют *алфавитной последовательностью*. При сортировке данных мы можем использовать то, что благодаря такой упорядоченности сохраняется алфавитный порядок литер. Упорядоченность специальных литер, а также относительный порядок трех групп литер (специальные литеры, буквы, цифры) в различных способах кодирования различаются. Поэтому можно писать программы, независимые от методов кодирования литер, если в них цепочки литер проверяются только на равенство. Проверка специальных литер на неравенство типа «меньше чем» делает программу зависимой от принятого варианта кодирования.

В ряде версий Фортрана нет данных «литерного» типа. Вместо этого закодированные литеры могут храниться в переменных

целого, вещественного или логического типа. Одна переменная может вместить несколько закодированных литер; сколько именно — зависит от принятого способа кодирования (BCD, ASCII или EBCDIC) и от длины машинного слова. Например, в машинах IBM 360-370 используется код EBCDIC (8 битов на литеру), а длина слова составляет 32 бита. Таким образом, в одном машинном слове могут быть закодированы четыре литеры. В машине DEC-10 приняты 36-разрядное слово и код ASCII (по 7 битов на литеру); таким образом, в одном слове можно уместить 5 литер. При этом крайний правый бит или «младший разряд» не используется. В машинах Univac 1100 принят код BCD (6 битов на литеру) и в 36-разрядном машинном слове можно уместить 6 литер.

3.3. Ввод и вывод литерных данных

Код формата A соответствует преобразованию литерных данных из внешней формы во внутреннее представление и обратно аналогично тому, как и код I обозначает преобразование во внутреннее представление и обратно целых величин. Код формата A записывается как Aw, w — размер поля. Предположим, что с помощью формата Aw передаются данные к (или от) переменной, в которой может храниться n закодированных литер. Если выполняется операция ввода и $w \leqslant n$, закодированные литеры занимают w левых позиций переменной, а правые $(n-w)$ позиций заполняются пробелами. Если $w \geqslant n$, только n правых литер входного поля умещаются в переменной. Если выполняется операция вывода и $w < n$, то на вывод передаются расположенные слева w литер. Если $w > n$, будут переданы все n литер, а левые $(w-n)$ позиций поля вывода заполняются пробелами.

Приведенный ниже пример поясняет эти правила. Предположим, что в используемой ЭВМ в одном машинном слове располагается пять литер. Выполнение следующей пары инструкций READ/FORMAT:

```
    READ (IN 100) INCH1, INCH2, INCH3
100 FORMAT (A3, A5, A8)
    КАРТА С ДАННЫМИ: CATZEBRADOGHOUSE
```

вызывает присваивание переменным следующих значений:

INCH1: CATbb INCH2: ZEBRA INCH3: HOUSE

(Результат выполнения показан в виде литер, а не в соответствующем двоичном представлении. Пробел отмечается буквой b.) Вывод содержимого указанных переменных по инструкции FORMAT с меткой 101 дает следующий результат:

```
      WRITE (IOUT, 101), INCH1, INCH2, INCH3
101   FORMAT (1X, 'OUTPUT=**', A4, A5, A8, '**')
OUTPUT=**CAT  ZEBRA  HOUSE**
```

На практике, чтобы специально не запоминать описанные выше правила ввода/вывода литер, мы предлагаем пользоваться следующими более удобными рекомендациями:

1. Если в машине, на которой вы работаете, можно уместить в машинном слове n литер, то используйте код формата Aw, где $w \leqslant n$. Если при выводе требуются дополнительные пробелы, задавайте их с помощью формата X.

2. Будьте последовательны: если ввод осуществляется по формату Aw, то и вывод следует осуществлять по формату Aw. Благодаря этому можно не беспокоиться о числе значащих литер и их позициях в переменной.

3. В различных применениях, там, где это возможно, многие программисты выбирают какую-то одну плотность упаковки в слове, скажем A4, для всех переменных, которые содержат литерные данные.

3.4. Литерные константы

Литерные константы могут быть представлены двумя способами. Во-первых, литерная константа может быть задана как цепочка литер с предшествующей буквой «H», перед которой указывается длина цепочки (H не принадлежит цепочке). При втором способе цепочка литер заключается в кавычки (апострофы). Если кавычка встретилась внутри цепочки литер, она повторяется дважды (в самой литерной константе останется только одна из двух кавычек). Из этих двух методов только первый является стандартным. Однако из-за неудобств и ошибок, связанных с подсчетом литер, в большинстве компиляторов реализован также и второй способ задания литерных констант. Ниже приведено несколько примеров литерных констант и цепочек литер, которые эти константы представляют:

```
КОНСТАНТЫ: '*' 4HPL/I  10H'TAIN'T SO '''TIS TOO'
ПРЕДСТАВЛЯЮТ: * PL/I  'TAIN'T SO 'TIS TOO
```

В стандарте Фортрана 1966 г. литерные константы были допустимы только в трех случаях: в инструкциях DATA, в инструкции FORMAT и как параметры подпрограмм. Однако в некоторых компиляторах разрешается использовать литерные константы и в других случаях. В Фортране 77 ограничения были значительно ослаблены (В разд. 3.8 приведена более детальная информация о работе с литерами в Фортране 77.)

3.5. Операции над литерами

После того как данные были занесены в переменные, над этими переменными можно выполнять арифметические и логические операции. Например, если STR1 и STR2 — две целые переменные, содержащие литерные данные, то инструкции

STR1=STR2 и IF(STR1 .EQ. STR2)

имеют смысл и записаны правильно. Так как код цепочки цифр не связан с представлением целого или вещественного числа, то инструкции

STR1=10. * STR2 или IF(STR1 .EQ. 10)

не имеют смысла и не дают ожидаемого результата. Кроме того, если одно и то же сочетание битов интерпретировать по-разному (как вещественное или как целое значение), то следующее сравнение будет ошибочным:

```
INTEGER ISTR/4HSAME/
REAL RSTR/4HSAME/
IF (ISTR .EQ. RSTR)
```

Хотя битовые представления ISTR и RSTR одинаковы, сравнение будет неправильным, поскольку интерпретация цепочки битов как целого числа отличается от ее же интерпретации как вещественного числа.

Опытные программисты хранят все литерные данные в переменных одного типа. Логическому типу свойственно ограничение, так как две логические переменные нельзя сравнить с помощью операции отношения («равно», «меньше чем» и т. д.). В большинстве ЭВМ операции над целыми числами (присваивание, сравнение) могут быть выполнены как минимум с той же скоростью, что и над вещественными, но обычно быстрее. Поэтому мы рекомендуем следующее правило:

> Литеры следует хранить только в переменных одного типа, предпочтительно в целых переменных.

3.6. Пример: бесформатный ввод целых

Вычислительный процесс обычно складывается из ввода данных и их обработки. Эти данные иногда целесообразно располагать в определенных фиксированных позициях перфокарты, однако чаще удобнее располагать их в произвольных позициях. Иметь такую возможность особенно ценно для начинающих программистов, которые, как правило, даже не представляют себе, сколь серьезными могут быть последствия, если данные будут отперфорированы со смещением хотя бы на одну колонку.

Для преобразования целого числа, введенного по формату I, необходимо выполнить следующие действия:

1. Найти и проанализировать последовательность литер в пределах определенного поля.
2. Вычислить двоичное значение, эквивалентное целому числу, представленному этими литерами.
3. Присвоить двоичное значение указанной переменной.

Желательно ослабить первое ограничение, а именно вместо того, чтобы считывать литеры из предопределенных колонок перфокарты, можно начать чтение с любой указанной программистом позиции и использовать первую же непрерывную цепочку литер на карте. Заметим, что форматные коды I и E в данном случае уже будут неподходящими, так как входная величина не располагается в фиксированных позициях перфокарты. В этом случае нам приходится самим делать бóльшую часть работы, которую обычно выполняет форматный процессор, преобразующий число, представленное на перфокарте, в эквивалентный двоичный код. Фактически перед нами та же задача, которая решается при бесформатном вводе/выводе. Вот шаги, которые необходимо выполнить:

1. Найти первую литеру числа, которое нужно преобразовать (т. е. первую отличную от пробела пробивку на карте).
2. Найти конец цепочки литер, представляющих преобразуемое число (т. е. найти первую отличную от цифры литеру после первой литеры, не являющейся пробелом, или конец карты, что встретится раньше).
3. Вычислить целое значение, представленное этой последовательностью литер.

Первый шаг не представляет трудностей. По существу, это последовательный просмотр таблицы; здесь элементы таблицы — это колонки вводной перфокарты. Будем искать признак числа — знак или цифру. Пробелы игнорируются, пока не будет найдена первая литера числа. Просмотр заканчивается неудачно, если литера не оказалась ни знаком, ни цифрой или если до конца перфокарты не встретилось ни одной литеры, отличной от пробела. В этом случае процесс прекращается и в качестве значения числа принимается ноль.

Второй шаг также легко выполнить. Все, что нужно, это проанализировать каждую литеру после первой. При этом могут возникнуть следующие три случая: найдена не цифра, достигнут конец перфокарты, число найденных цифр больше допустимого. В первом случае анализ цепочки литер завершается нормально. Положение отличного от цифры разделителя необходимо заметить для дальнейшего последовательного просмотра карты.

Во втором случае мы имеем дело с другой формой нормального окончания. Этот факт также следует заметить; он означает, что поиск числа на данной перфокарте завершен. В третьем случае число задано ошибочно. Существует фиксированный, максимально допустимый размер целого числа. Этот размер различен для различных ЭВМ и зависит от длины слова машины. Для представления целых чисел будем пользоваться девятью десятичными цифрами, поскольку такие числа можно расположить в одном машинном слове в большинстве больших вычислительных машин.

Третий шаг легко выполнить в две фазы. Во-первых, необходимо представить каждую цифровую литеру эквивалентным двоичным целым и, во-вторых, объединить отдельные двоичные целые в одно число. Первая фаза просто добавляется к просмотру таблицы: литере «0» ставим в соответствие число 0; литере «1» — целое 1 и т. д. Заметим, что так как литера «1» и целое 1 имеют разные представления внутри машины, то необходимо самим позаботиться о соответствии. (Существует взаимосвязь между представлениями, но она зависит от машины.)

Соответствие между литерой и целым может быть получено с помощью двумерного массива, один столбец которого содержит цифру в форме литеры от '0' до '9', второй — соответствующее литере целое число.

Литера	Представление
'0'	0
'1'	1
...	...
'9'	9

Заметим, что каждое значение во втором столбце этой таблицы соответствует порядковому номеру строки минус 1. Таким образом, второй столбец не нужен, поскольку его элементы можно легко вычислить: достаточно вычесть 1 из соответствующего значения индекса. В приведенной ниже программе будет использован этот способ.

Теперь рассмотрим вторую фазу последнего шага, т. е. объединение целых чисел, соответствующих цифрам некоторого числа. Пусть число представлено цепочкой литер C1 C2 . . . Cn, и пусть $i1$, $i2$, . . ., in будут целые числа, соответствующие литерам C1, C2, . . ., Cn. Значение этой цепочки равно

$$i_n + 10 * i_{n-1} + 100 * i_{n-2} + \ldots + 10^{n-2} * i_2 + 10^{n-1} * i_1.$$

Его можно вычислить справа налево умножением очередного i_k на соответствующую степень 10. Более удобная форма записи для вычисления этой формулы

$$10^{n-1} * i_1 + 10^{n-2} * i_2 + \ldots + 10^2 * i_{n-2} + 10^1 * i_{n-1} + i_n,$$

или, после перегруппировки умножений,

$$((\ldots(((i_1)*10+i_2)*10+i_3)\ldots*10+i_{n-1})*10+i_n).$$

Вычисление последней формулы может быть организовано в цикле:

1. TERM=0;
2. DO=ИНДЕКС=1 до N:
   TERM=TERM*10+I (ИНДЕКС)

Этот алгоритм имеет несколько преимуществ перед приведенным выше:

1. Цифры просматриваются слева направо, что делает программу более понятной.
2. Нет необходимости производить предварительный просмотр данных для определения правого конца цепочки цифр.
3. Требуется выполнить только n умножений, в то время как в предыдущей формуле для вычисления каждого множителя $10**k$ требуется до k умножений или возведений в степень (в зависимости от выбранного метода). В свою очередь множитель $10**k$ затем необходимо умножить на I (k). Таким образом, как минимум требуется $2*n$ умножений или возведений в степень. Обе эти операции требуют машинного времени, и от них следует попытаться избавиться.

Все это не пустые хлопоты. В гл. 1 было показано, что с точки зрения эффективности лучше выбрать хороший алгоритм и реализовать его в незамысловатой манере, чем пытаться компенсировать недостатки алгоритма хитроумными приемами. Программа, реализующая только что описанный алгоритм, приведена на рис. 3.1.

Эта программа не требует подробного обсуждения, так как она просто отражает уже принятые нами решения на уровне алгоритма. Заметим только, что единственное указание о том, что длина записи равна 80, содержится в первой инструкции инициализации данных. Во всех последующих инструкциях используется значение переменной ENDCOL. Если позже потребуются записи, занимающие только первые 72 колонки на перфокарте, будет легко внести изменение в программу. При этом не потребуется просматривать всю программу в поисках инструкций, содержащих число 80, или возиться с отладкой, если один раз это число было упущено. Значение последней колонки можно также передавать как значение параметра. Заметим, что поиск чисел проводится в отдельной подпрограмме DFIND. Отчасти это сделано для большей наглядности, отчасти — потому, что этот модуль может быть использован еще где-либо. Сформулируем общие требования, предъявляемые к написанию программы:

```
      SUBROUTINE EDCHIN (CARD, COL, INT, ERFLAG)
C * * EDCHIN: * * ПРЕОБРАЗОВАНИЕ ЛИТЕР В ЦЕЛЫЕ * * * * * * * * * * S
C
C   ВХОД:
C     CARD:     МАССИВ ИЗ 80 ЛИТЕР, ПРЕДСТАВЛЯЮЩИХ КОЛОНКИ ПЕР-
C               ФОКАРТЫ (ПО ОДНОЙ ЛИТЕРЕ НА ЭЛЕМЕНТ);
C     COL:      ПОЗИЦИЯ ВНУТРИ «CARD», С КОТОРОЙ НАЧИНАЕТСЯ ПОИСК
C               ЦЕПОЧКИ ЦИФР.
C
C   ВЫХОД:
C     COL:      ПОЗИЦИЯ ВНУТРИ «CARD», НА КОТОРОЙ ПРЕКРАТИЛСЯ
C               ДАЛЬНЕЙШИЙ ПОИСК ЦИФР;
C     A.        COL=81; ЕСЛИ НЕ НАЙДЕН СИМВОЛ, ОТЛИЧНЫЙ ОТ ЦИФРЫ,
C               ПРЕКРАТИТЬ ПОИСК ПОСЛЕ АНАЛИЗА 80-Й КОЛОНКИ;
C     B.        COL=ПОЗИЦИЯ ЛИТЕРЫ ПОСЛЕ ПОСЛЕДНЕЙ НАЙДЕННОЙ
C               ЦИФРЫ;
C     INT:      ДВОИЧНОЕ ЗНАЧЕНИЕ, ЭКВИВАЛЕНТНОЕ НАЙДЕННОЙ ЦЕ-
C               ПОЧКЕ ЦИФР;
C     A.        INT=0, ЕСЛИ НЕ ВСТРЕТИЛОСЬ ЦИФРЫ ДО ПЕРВОЙ НЕДОЗ-
C               ВОЛЕННОЙ ЛИТЕРЫ;
C     B.        INT=ЗНАЧЕНИЕ, СООТВЕТСТВУЮЩЕЕ ПЕРВЫМ ДЕВЯТИ
C               ЦИФРАМ, ЕСЛИ НАЙДЕНА ЦЕПОЧКА ДЛИНЕЕ ДЕВЯТИ ЦИФР;
C     C.        INT=ЦЕЛОЕ ЧИСЛО, ЕСЛИ НАЙДЕНА ВЕРНАЯ ЦЕПОЧКА;
C     ERFLAG:   ФЛАГ, СИГНАЛИЗИРУЮЩИЙ ОБ ОШИБКЕ;
C     A.        ERFLAG=—1, ЕСЛИ НЕ НАЙДЕНО НИ ОДНОЙ ЦИФРЫ;
C     B.        ERFLAG=0. ЕСЛИ ЦЕЛОЕ НАЙДЕНО И ПРЕОБРАЗОВАНО;
C     C.        ERFLAG=+1, ЕСЛИ НАЙДЕНО БОЛЕЕ 9 ДЕСЯТИЧНЫХ ЦИФР.
C   МЕТОД:
C     АНАЛИЗИРУЕТСЯ ЭЛЕМЕНТ ЗА ЭЛЕМЕНТОМ МАССИВА. ПРИ ЭТОМ
C     ИЩУТСЯ:
C     A.        ОДИН ИЛИ БОЛЕЕ ПРОБЕЛОВ, КОТОРЫЕ ИГНОРИРУЮТСЯ;
C     B.        ЗНАК (+ ИЛИ —), КОТОРЫЙ СЛУЖИТ ЗНАКОМ РЕЗУЛЬТИРУ-
C               ЮЩЕГО ЦЕЛОГО ЧИСЛА;
C     C.        ЦИФРА, С КОТОРОЙ НАЧИНАЕТСЯ ПОИСК НЕПРЕРЫВНОЙ ЦЕ-
C               ПОЧКИ ЦРФР;
C     D.        НЕДОЗВОЛЕННАЯ ЛИТЕРА, ОСТАНАВЛИВАЮЩАЯ ВЫПОЛНЕ-
C               НИЕ ПОДПРОГРАММЫ.
C     КАК ТОЛЬКО НАЙДЕНА ПЕРВАЯ ЦИФРА, ЗА НЕЙ ДОЛЖНЫ СЛЕДОВАТЬ
C               ОСТАЛЬНЫЕ ЦИФРЫ; ПРОБЕЛЫ И ЗНАКИ НЕ РАЗРЕШАЮТСЯ.
C               ИСПОЛЬЗУЕТСЯ МАКСИМАЛЬНО 9 ЦИФР.
C
C     ПРОГРАММИСТ: ———          ДАТА: ———
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *F
      INTEGER CARD(80),COL,INT,ENDCOL,ERFLAG
      INTEGER PLUS,MINUS,BLANK,SIGN,END,DFIND
      DATA PLUS/'+'/,MINUS/'-'/,BLANK/' '/,ENDCOL/80/
      SIGN=+1
      ERFLAG=-1
      INT=0
C       ДОПУСТИМО ЛИ НАЧАЛЬНОЕ ЗНАЧЕНИЕ COL?
      IF (COL .GT. ENDCOL) GO TO 99
C
C       НАЧАЛО ПОИСКА ПЕРВОЙ ЦИФРЫ
   10 DO 20 I=COL,ENDCOL
         IF (CARD(I) .EQ. MINUS) GO TO 30
         IF (CARD(I) .EQ. PLUS) GO TO 35
         IF (CARD(I) .NE. BLANK) GO TO 40
C       ЕСЛИ НЕ ПРОБЕЛ ИЛИ ЗНАК, ТО ИЛИ ЦИФРА
C       ИЛИ НЕВЕРНАЯ ЛИТЕРА, РЕШИМ ПОЗДНЕЕ
   20 CONTINUE
C       ЕСЛИ ВЫШЛИ СЮДА, ВСЕ КОЛОНКИ БЫЛИ ЗАПОЛНЕНЫ ПРОБЕЛАМИ,
C       ВОЗВРАТ ПО ОШИБКЕ
      COL = ENDCOL+1
      RETURN
C
C       ЗДЕСЬ ОБНАРУЖЕН ЗНАК МИНУС
   30 SIGN = -1
C       ЗДЕСЬ ОБНАРУЖЕН ЗНАК ПЛЮС ИЛИ МИНУС.
C       ЕСЛИ ПЛЮС, „SIGN“ НА ВХОДЕ БЫЛ УСТАНОВЛЕН ПРАВИЛЬНО
C       В ЛЮБОМ СЛУЧАЕ ПЕРЕЙТИ К СЛЕДУЮЩЕЙ КОЛОНКЕ
   35 I = I+1
C       УБЕДИМСЯ, ЧТО В ПОСЛЕДНЕЙ КОЛОНКЕ НЕТ ЗНАКА
```

Рис. 3.1. Процедура преобразования литер в целые числа.

```
      IF (I .GT. ENDCOL) RETURN
C
C           НАЧАЛО ПОИСКА ЦЕПОЧКИ ЦИФР (НЕ БОЛЕЕ 9)
   40 END = MINO (ENDCOL, I+8)
      DO 50 J=I,END
C           ВЫЗОВ ФУНКЦИИ ПОИСКА, ОНА ВОЗВРАЩАЕТ
C           ЦЕЛЫЕ ОТ 0 ДО 9, ЕСЛИ ЦИФРА НАЙДЕНА,
C           И -1, ЕСЛИ НАЙДЕНА НЕ-ЦИФРА
         M = DFIND (CARD(J))
C           ЕСЛИ НЕ-ЦИФРА, ОСТАНОВ; ИНАЧЕ ЗАПОМНИТЬ ЗНАЧЕНИЕ
C           И ПРОДОЛЖАТЬ ОБРАБОТКУ
         IF (M .LT. 0)  GO TO 60
         INT = 10*INT + M
         ERFLAG=0
   50 CONTINUE
C
C           ЕСЛИ ВЫШЛИ СЮДА, ТО НАЙДЕН КОНЕЦ КАРТЫ ИЛИ 9 ЦИФР;
C           ПЕРЕДВИНУТЬСЯ К НЕ-ЦИФРЕ ИЛИ КОНЦУ КАРТЫ
      IF (END .LT. ENDCOL) GO TO 55
C           СЛУЧАЙ 1: КОНЕЦ КАРТЫ; ОСТАНОВ
         J = ENDCOL+1
         GO TO 60
C
C  ЕСЛИ (END .LT. ENDCOL)
C           СЛУЧАЙ 2: 9 ЦИФР ПРЕОБРАЗОВАНЫ; НАХОДИМ СЛЕД. НЕ-ЦИФРУ
   55 END = END+1
      J = END
C           ЕСЛИ ДЕСЯТАЯ ЛИТЕРА НЕ-ЦИФРА, ЦЕПОЧКА ПРИЕМЛЕМА
      IF (DFIND(CARD(J) .EQ. -1) GO TO 60
      ERFLAG=+1
      DO 58 J = END,ENDCOL
         IF (DFIND(CARD(J)) .LT. 0) GO TO 60
   58 CONTINUE
C           ВЫШЛИ НА КОНЕЦ КАРТЫ
      J = ENDCOL+1
C
C           ВОЗВРАЩЕНИЕ ВЫЧИСЛЕННОГО ЦЕЛОГО ЗНАЧЕНИЯ
   60 INT = INT*SIGN
      COL = J
      RETURN
C
C           НЕВЕРНОЕ ЗНАЧЕНИЕ "COL"; ВОЗВРАЩАЕМ 0 И
C           COL = LAST-CARD-COLUMN + 1
   99 INT = 0
      COL = ENDCOL+1
      ERFLAG = -1
      RETURN
      END

      INTEGER FUNCTION DFIND (CHAR)
C
C     ПРИНИМАЕТ НА ВХОДЕ ОДИНОЧНУЮ ЛИТЕРУ, ОПРЕДЕЛЯЕТ,
C     ЦИФРА ЛИ ЭТО И ВОЗВРАЩАЕТ ЦЕЛОЕ,
C     ЭКВИВАЛЕНТНОЕ ЭТОЙ ЦИФРОВОЙ ЛИТЕРЕ.
C     ЕСЛИ ЛИТЕРА ОТЛИЧНА ОТ ЦИФРЫ, ПОДПРОГРАММА ВОЗВРАЩАЕТ -1
C     ДЕЙСТВИЕ - ПОИСК В ТАБЛИЦЕ „DIGIT".
C     ЛИТЕРЕ '0' СООТВ. 1-ЫЙ ЭЛЕМ. ТАБЛИЦЫ, ЛИТЕРЕ '1' - 2-ОЙ И Т.Д.
C     ЕСЛИ ДЛЯ ЛИТЕРЫ ЕСТЬ ЭЛЕМЕНТ В ТАБЛИЦЕ,
C     ЕЕ ЦЕЛОЕ ЗНАЧЕНИЕ НА 1 БОЛЬШЕ,
C     ЧЕМ ЕЕ ПОЗИЦИЯ В ТАБЛИЦЕ
C
      INTEGER CHAR, DIGIT(10)
      DATA DIGIT/'0','1','2','3','4','5','6','7','8','9'/
C           ВНАЧАЛЕ БЕРЕМ ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ
      DFIND = -1
C
C           ПОИСК ЛИТЕРЫ;
C           ЕСЛИ НАЙДЕНА, ЗАМЕТИМ ЕЕ ПОЗИЦИЮ
      DO 10 I = 1, 10
         IF (CHAR .EQ. DIGIT(I)) DFIND = I-1
   10 CONTINUE
      RETURN
      END
```

Продолжение Рис. 3.1.

1. Программируя, в пределах разумного стремитесь к общности (не делайте, однако, фетиша из структуры и эффективности).

2. Планируйте возможные модификации программы. Предусмотрите вероятные изменения в программе так, чтобы вы или еще кто-либо мог произвести эти изменения. Выделите в отдельные модули те части, которые могут быть использованы в каких-то других программах.

3.7. Другие методы работы с литерами

В этом разделе мы рассмотрим методы выполнения более сложных операций над литерами. Для простоты будем считать, что в машинном слове располагается одна литера. В первую очередь проанализируем операции конкатенации (объединения двух цепочек литер) и поиска подцепочек.

Для представления цепочки литер используется одномерный, а для массива цепочек — двумерный массив. Предположим, что мы имеем дело с массивом, содержащим не более 200 имен, каждое длиной не более 25 литер. Тогда выделить память под эти литеры можно посредством декларации.

```
INTEGER NAME (25, 200)
```

Заметим, что в большинстве случаев не имеет значения, как задать массив: 25*200 или 200*25. Однако, учитывая, что в Фортране массивы хранятся по столбцам и большинство обращений к массиву происходит одновременно по всем литерам одного имени, целесообразно хранить литеры в такой последовательности, чтобы доступ к ним был удобным. Поэтому форму, примененную в приведенной выше инструкции, следует предпочесть.

Чтобы прочитать N-е имя при заданном значении N, выполним инструкцию

```
READ (—,—) (NAME(I, N)=1,25)
```

Аналогично, чтобы проверить, являются ли K-е и (K+1)-е имена одинаковыми, выполним инструкции

```
      DO 50 J = 1,25
         IF (NAME(J,K) .NE. NAME(J,K+1)) GO TO 400
   50 CONTINUE
C              ИМЕНА СОВПАДАЮТ, ЕСЛИ ЦИКЛ
C              ЗАВЕРШИЛСЯ НОРМАЛЬНО
```

Процесс конкатенации не такой уж простой. Чтобы объединить две цепочки литер, необходимо иметь достаточно большой массив, в котором могли бы разместиться обе цепочки, и затем нужно переслать их в этот массив так, чтобы в массиве они ока-

зались рядом. Пересылка должна выполняться поэлементно. Предположим, что требуется конкатенировать две цепочки литер STRA и STRB. Длины цепочек STRA и STRB хранятся в переменных LENA и LENB соответственно. Если цепочка, полученная в результате их конкатенации, должна храниться в массиве STRC, а ее длина — в переменной LENC, то для конкатенации цепочек можно выполнить следующий фрагмент программы:

```
      IF (LENA) 99,    20,      11
C                ОШИБКА, НЕТ STRA, НОРМАЛЬНО
   11    DO 15 I = 1, LENA
           STRC(I) = STRA(I)
   15    CONTINUE
   20 IF (LENB) 99,    40,      21
C                ОШИБКА, НЕТ STRB, НОРМАЛЬНО
   21    DO 25 I = 1, LENB
           STRC(I+LENA) = STRB(I)
   25    CONTINUE
   40 LENC = LENA + LENB
      ...
C                ОШИБКА: LENA<0 ИЛИ LENB<0
   99 ...
```

Важным является значение длины, хранящееся в памяти вместе с каждой цепочкой литер. Длина должна быть обязательно целой переменной точно так же, как и элементы цепочки литер. Для удобства отладки и локализации обращений к памяти в процессе выполнения длину цепочки можно хранить в первом элементе массива, а сами литеры расположить во втором и последующих элементах. Так, цепочка литер 'HELLO' длиной 5 может быть представлена в целом массиве A из 6 элементов:

5	'H'	'E'	'L'	'L'	'O'

Заметим, что для наглядности программы может оказаться желательным использовать различные имена для задания самой цепочки литер и ее длины, как STRA и LENA в приведенном выше примере. Это удобно сделать с помощью инструкции EQUIVALENCE следующим образом:

```
INTEGER A(101), LENA, STRA(100)
EQUIVALENCE (A(1),LENA), (A(2),STRA(1))
```

При такой организации легко выполняется отладка программы, поскольку вся информация о цепочке «A» содержится в одном наборе ячеек памяти, которые могут быть выведены в виде дампа посредством одной инструкции.

Другим важным средством при работе с литерами являются операции над подцепочками. Подцепочка — это любая непрерывная часть цепочки, включая саму цепочку. Одной из типичных задач при работе с литерами может быть поиск определенного вида подцепочек, например только пробелов или наибольшей подцепочки, которая не содержит литеру «?». Алгоритм решения таких задач довольно очевиден, и далее мы его не обсуждаем.

Рассмотрим теперь такие действия с подцепочками, как присваивание (присваивание переменной определенной подцепочки другой переменной) и замещение (замена одной подцепочки переменной на другую). Метод выполнения операции замещения будет проиллюстрирован в примере, приведенном в конце данной главы. Здесь же мы рассмотрим, как одной переменной присвоить подцепочку другой переменной.

Подцепочки обычно определяют так: подцепочка цепочки X, начинающаяся с литеры I, длиной J литер с соответствующими значениями I и J. Предположим, что цепочки литер являются массивами, имеют переменную длину и состоят из последовательности элементов, каждый из которых содержит одну литеру. Наша задача — присвоить цепочке DEST подцепочку цепочки SOURCE, начиная с позиции POS длиной, заданной в переменной SUBLEN. Будем полагать, что длины цепочек DEST и SOURCE указаны в переменных DSTLEN и SRCLEN соответственно. Это можно сделать с помощью следующего набора инструкций:

```
      IF((POS+SUBLEN-1) .GT. SRCLEN) GO TO 999
C ОШИБКА: ЦЕПОЧКА КОРОЧЕ ТРЕБУЕМОЙ ПОДЦЕПОЧКИ
C
      DO 10 I=1,SUBLEN
         DEST(I) = SOURCE(POS+I-1)
 10   CONTINUE
      DSTLEN = SUBLEN
```

Программы, в которых для хранения одной литеры отводится переменная целого типа, имеют одно очевидное достоинство: такие программы правильно выполняются на любых ЭВМ, в которых допускается как минимум полный стандарт Фортрана ANS 1966 г. Кроме того, можно использовать любую легко доступную машину при отладке программ, предназначенных для малой ЭВМ, доступ к которой ограничен. Можно также писать программы, предназначенные для распространения среди пользователей на различных ЭВМ. Этим преимуществом обладают немногие другие языки. Однако, как мы увидим в следующем разделе, такой метод сопряжен с довольно громоздким программированием.

3.8. Данные литерного типа в языках WATFIV и Фортран 77

Рассмотренные нами в предыдущем разделе методы работы с литерами требуют большого количества инструкций для выполнения довольно простых операций. Только для того, чтобы переслать цепочку литер, необходимо организовать цикл и выполнить много ссылок на переменные с индексами. Простые и наглядные сравнения (например, IF(X ·EQ. 'HOUSE')) или присваивания (Y='ALPHA') недопустимы. В историческом плане эти обстоятельства понятны, так как в начальном проекте языка Фортран средства работы с литерами не были предусмотрены. Компиляторы, допускающие лишь те немногие возможности обработки литер, которые были описаны выше, в течение какого-то времени все еще будут общепринятыми, и, чтобы программы были портативными, приходится ограничивать средства работы с литерами. Однако уже и сейчас есть компиляторы, которые располагают средствами для более естественной работы с литерами. Среди таких компиляторов одним из наиболее распространенных является учебно-ориентированный компилятор WATFIV Университета Ватерлоо. Действительно, средства обработки литер этого компилятора настолько разумны, что они практически без изменения были перенесены в Фортран 77. В компиляторе WATFIV добавлены литерные данные типа CHARACTER. Посредством декларации CHARACTER цепочки литер могут быть описаны аналогично тому, как целые или вещественные числа объявляются в декларациях INTEGER или REAL. Декларация CHARACTER указывает, что определенные переменные являются переменными литерного типа. Она позволяет также указать длину каждой переменной. Длина задается с помощью константы, которой предшествует *. Длина указывается вслед за словом CHARACTER (в этом случае она относится ко всем переменным, содержащимся в данной инструкции) или после имени переменной (и тогда она относится только к данной переменной, т. е. перекрывает длину, заданную после слова CHARACTER). Если в декларации CHARACTER длина не задана, она полагается равной 1. Примеры описания литерных переменных приведены ниже:

```
CHARACTER*5 STRG1, STRG2
CHARACTER NAME*20, ADDR*30, CITY*23, STATE*2, ZIP*5
CHARACTER    LETR, ALPH*26, DIGIT
```

В первом примере декларируются две литерные переменные длиной по 5 литер каждая. Во втором примере декларируется 5 переменных, каждая из которых содержит соответственно 20, 30, 23, 2 и 5 литер. В последней декларации переменные LETR

и DIGIT содержат по одной литере, а переменная ALPH — 26 литер.

Аналогично, как и в случае численных массивов, могут быть объявлены массивы литерных переменных. Например, в следующей декларации объявляется массив NAME из 100 элементов, где каждый элемент представляет собой цепочку из 20 литер:

```
CHARACTER*20 NAME (100)
```

3.8.1. Литерные константы

При рассмотрении литерных данных естественно возникает вопрос о литерных константах и операциях над ними. Определение литерных констант было дано выше — это цепочка литер, заключенная в апострофы. Литера «апостроф» в такой цепочке представляется двумя последовательными апострофами. В отличие от ранее рассмотренных возможностей компилятор WATFIV позволяет задавать литерные константы и другим образом. Как и раньше, литерная константа может встретиться в инструкции FORMAT, в инструкции присваивания начальных значений DATA, в качестве параметра подпрограммы. Кроме того, литерная константа может выступать в качестве правой части инструкции присваивания. Она может быть также операндом для операции отношения в логической инструкции IF. Таким образом, в компиляторе WATFIV инструкции типа IF (STR1 .LT. 'A') и STR2='ABC' допустимы.

Как и ранее, значение цепочки литер отличается от значения целой или вещественной константы или имени переменной. Так, инструкции вида IF ('500' .EQ. 500) или A='A+1' могут быть синтаксически допустимы для некоторых компиляторов, но их смысл отличается от того, что может показаться с первого взгляда.

Переменная, описанная в декларации CHARACTER*n, в качестве значения будет иметь цепочку ровно из n литер. (Если переменной не присвоено значения, то оно, естественно, не предсказуемо.)

Если переменной, описанной в инструкции CHARACTER*n, присваивается значение, длина которого n, то будут использованы только n левых литер. Если в значении переменной литер меньше, чем n, то оставшиеся правые позиции переменной заполняются пробелами. Это же правило распространяется и на константы. Пусть STR4A и STR4B — переменные, описанные как CHARACTER*4. Тогда каждая из инструкций

```
STR4A = 'ABCDEF'          STR4B = 'XYZ'
STR4A = 'ABCD'            STR4B = 'XYZ '
```

присваивает переменным STR4A (соответственно STR4B) одинаковые значения.

При операции сравнения более короткий операнд дополняется справа пробелами до длины более длинного операнда.

Переменная типа CHARACTER*n представляет собой неразделимый блок из n литер. В результате выполнения операции присваивания происходит замещение, а при выполнении операции сравнения — сопоставление всех n литер (включая пробелы, добавленные для удлиннения короткого операнда). Для анализа или изменения одиночной литеры из переменной типа CHARACTER*n можно также использовать инструкцию EQUIVALENCE.

Рассмотрим переменную CITY типа CHARACTER*20, представляющую собой блок из 20 литер, используемый при сортировке в алфавитном порядке. Предположим, что мы задались целью напечатать название города, т. е. только значащую часть цепочки CITY, игнорируя пробелы справа, дополняющие ее до 20 литер. Обнаружить, где заканчиваются отличные от пробелов литеры цепочки, можно с помощью следующего набора инструкций:

```
      CHARACTER*20 CITY
      CHARACTER*1 C1(20)
      EQUIVALENCE (CITY, C1(1))
      DO 10 I = 1, 20
         IF (C1(21-I) .NE. ' ') GO TO 20
   10 CONTINUE
C        +++ ОШИБКА—ЦЕПОЧКА СОСТОИТ ИЗ ПРОБЕЛОВ
         :
         :
C        ... НАЙДЕНА ПОСЛЕДНЯЯ ЛИТЕРА—„НЕ ПРОБЕЛ"
   20 LNBLNK = 21-I
```

Таким образом мы можем узнать точную длину значащей части переменной CITY (в данном случае названия города; CITY — это город). В данном примере используется инструкция эквивалентности, которая позволяет установить начало переменных CITY и C1 в одну и ту же точку памяти;

(1) переменная CITY занимает 20 последовательных литер (байтов) в памяти;

(2) переменная C1 представляет массив из 20 элементов; каждому элементу отводится один байт.

Ясно, что, поскольку переменные CITY и C1 ассоциированы с одним и тем же адресом памяти, значение одной переменной является значением другой переменной. Однако каждый элемент массива типа CHARACTER*1 представляет собой одиночную литеру из цепочки CITY. Таким образом, мы получаем сразу две прекрасные возможности: можем обрабатывать одновременно 20 литер под именем CITY и оперировать с любой одиночной литерой из этой цепочки как с элементом массива C1.

3.9. Примеры

Мы завершаем эту главу двумя служебными программами. Первую программу можно использовать для создания документа, описывающего каждый отдельный модуль программы. Вторую — для преобразования исходного текста программы так, чтобы яснее была видна последовательность действий. Вторая программа будет выполнять такие функции, как выделение (с помощью отступа) тел DO-циклов и изменение имен переменных. Обе программы широко применяют средства работы с литерами.

В первом примере мы построим программу, которая создает документ, описывающий логику работы других программ (Program Logic Manual — PLM). Назначение и содержание такого документа мы объяснили в гл. 1. Для каждого модуля программы будет построена секция, содержащая описание этого модуля: назначение модуля (в общих чертах), его алгоритм; используемые имена, параметры и наиболее важные переменные; условия входа и выхода; имена модулей, вызываемых данным модулем, и имена модулей, вызывающих данный модуль.

В полном объеме эту информацию невозможно «выудить» из программы. Однако точно такую информацию, мы обычно описываем в начале каждого модуля в «шапках» с комментариями. Такая информация составляет важную часть исходного текста программы, она полезна при отладке и работе с программой. Кроме того, в такой форме комментарии легко найти при анализе программы. Мы разработаем программу, которая принимает исходный текст другой программы и выуживает из него комментарии для построения документа PLM.

Прежде всего нам необходимо выбрать форму, которую должны иметь «шапки» с комментариями. Нужно выбрать только те комментарии, которые действительно содержат описание модуля. Эта форма, конечно, произвольна, поэтому можно принять, например, такую форму:

```
C * * ИМЯ (ПАРАМ1, ПАРАМ2, . . ., ПАРАМN) * * * * * * * * * * * *
C     ПОДПРОГРАММА «ИМЯ»                                          *
C     УСЛОВИЯ ВХОДА . . .                                         *
C     УСЛОВИЯ ВЫХОДА . . . И Т. Д.                                *
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * F
```

В первой строке ИМЯ — это имя модуля, а ПАРАМ — символические параметры. В том случае, если данный модуль представляет собой главную программу, первая строка будет заполнена чередующимися звездочками и пробелами. В колонке 71 будет пробита буква M (начало главной программы), S (начало подпрограммы) или E (точка входа в подпрограмму). Блок с комментариями будет оканчиваться строкой такого вида: C * * в колонках с 1 по 6 и буквой F (финиш) в колонке 71.

1. Прочитать перфокарту. Если перфокарт больше не осталось, останов.
2. В колонках с 1 по 6 'C * *'? Нет, перейти к шагу 1.
3. В колонке 72 буква 'F'? Да, перейти к шагу 1.
4. Заменить *имя* на содержимое колонок с 7 по 12.
5. В колонке 72 буква 'M'? Если да, то присвоить *имени* значение 'M/PROG'.
6. В колонке 72 буква 'M', 'S' или на текущей странице осталось менее 5 строк? Если да, перейти на новую страницу.
7. Распечатать колонки перфокарты с 1 по 72, заменяя содержимое первой и 72-й колонок пробелами.
8. Прочитать перфокарту; если карт не осталось, останов.
9. Колонки с 1 по 6 содержат 'C * *'? Да, перейти к шагу 1. Нет, осталась ли на текущей странице еще хотя бы одна строка? Нет, перейти на новую страницу.
10. Колонка 1=' ' , колонка 72=' ' .
11. Перейти к шагу 7.

Заметим, что в рассмотренной схеме модуля широко используются конструкции IF-THEN, IF-THEN-ELSE. Мы выделим секцию вывода в отдельную подпрограмму. По двум причинам. Прежде всего потому, что модуль вывода можно написать сначала в тривиальной форме (одна инструкция WRITE и RETURN) только для проверки остальной части программы, а затем, когда дела пойдут хорошо, можно побеспокоиться об эстетическом оформлении вывода. Во-вторых, подпрограмма вывода представляет собой отделимое тело текста программы. Удаляя его из главной программы, мы тем самым сокращаем ее объем и одновременно делаем программу более понятной. Как мы отмечали в гл. 1, следует стремиться к тому, чтобы подпрограммы были достаточно короткими и текст самой большой секции в идеальном случае умещался на отдельной странице листинга. Одним из путей сокращения размеров модуля является разумное выделение подпрограмм.

Программа, в которой реализован изложенный алгоритм, приведена на рис. 3.2. Часть вывода этой программы иллюстрируется на рис. 3.3. При выполнении программа создает документацию о самой себе, т. е. в качестве входа используется абсолютно идентичная программа.

В последнем примере этой главы мы рассмотрим программу, преобразующую форматы других программ. Напомним некоторые положения гл. 1: текст программы легче читать, если он начинается в соответствующих колонках; циклы легче проследить если тело цикла выделено; текст программы легче понять, если имена переменных имеют определенный смысл. Однако не все

```
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * M
C
C  ПРОГРАММА ПОСТРОЕНИЯ PLM
C  (ДОКУМЕНТОВ ПО ЛОГИКЕ РАБОТЫ ПРОГРАММ)
C
C  ПРОГРАММА ПРИНИМАЕТ ТЕКСТ ДРУГОЙ ПРОГРАММЫ НА ИСХОДНОМ
C  ЯЗЫКЕ И ИЗВЛЕКАЕТ БЛОКИ КОММЕНТАРИЕВ ИЗ КАЖДОЙ ПОДПРО-
C   ГРАММЫ ДЛЯ СОЗДАНИЯ PLM
C
C  ЛОГИКА РАБОТЫ:
C    ВВОДЯТСЯ КАРТЫ И ОСУЩЕСТВЛЯЕТСЯ ПОИСК ПОСЛЕДОВАТЕЛЬНО-
C    СТИ C * * В КОЛОНКАХ 1—6. ЕСЛИ ТАКАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ
C    НАЙДЕНА, ТО ЭТО ОТМЕЧАЕТ НАЧАЛО БЛОКА КОММЕНТАРИЕВ. АНА-
C    ЛИЗИРУЕТСЯ КОЛОНКА 72 ДЛЯ ОПРЕДЕЛЕНИЯ:
C                ГЛАВНОЙ ПРОГРАММЫ (КОЛ. 72='M')
C                ПОДПРОГРАММЫ (КОЛ. 72='S')
C                ТОЧКИ ВХОДА (КОЛ. 72='E')
C    ЕСЛИ ЭТО ГЛАВНАЯ ПРОГРАММА, ТО МОДУЛЬ БУДЕТ НАЗЫВАТЬСЯ
C                'M/PROG'.
C    ЕСЛИ ЭТО ПОДПРОГРАММА ИЛИ ТОЧКА ВХОДА, ТО ИМЕНЕМ СТАНО-
C                ВИТСЯ ЦЕПОЧКА БУКВ, СЧИТАННАЯ В КОЛОНКАХ 7—12.
C    ЭТО ИМЯ ПЕЧАТАЕТСЯ В ВЕРХНЕЙ ЧАСТИ КАЖДОЙ СТРАНИЦЫ PLM.
C                ЛИСТИНГ ИНСТРУКЦИЙ ПРОДОЛЖАЕТСЯ В PLM ДО ТЕХ
C    ПОР, ПОКА НЕ БУДЕТ СЧИТАНА СЛЕДУЮЩАЯ КАРТА, СОДЕРЖАЩАЯ
C                'C * *' В КОЛОНКАХ 1—6 И 'F' В КОЛОНКЕ 72.
C    ЗАТЕМ ПРОДОЛЖАЕТСЯ ПОИСК ПЕРВОЙ ПЕРФОКАРТЫ БЛОКА КОМ-
C                МЕНТАРИЕВ НОВОГО ПРОГРАММНОГО МОДУЛЯ.
C
C  ОСНОВНЫЕ ПЕРЕМЕННЫЕ:
C    NAME:       ИМЯ МОДУЛЯ (ОСНОВНОЙ ПРОГРАММЫ, ПОДПРОГРАММЫ
C                ИЛИ ТОЧКИ ВХОДА);
C    LINE;       ТОЛЬКО ЧТО СЧИТАННАЯ КАРТА. ПОСРЕДСТВОМ ЭКВИВА-
C                ЛЕНТНОСТИ ПОЭТОМУ МОЖНО ССЫЛАТЬСЯ НА:
C                COL:CHARACTER*1      МАССИВ ИЗ 72 ЭЛЕМЕНТОВ;
C                C1TO6:CHARACTER*6    ПЕРЕМЕННАЯ, СОДЕРЖАЩАЯ
C                                     КОЛОНКИ 1—6;
C                C7TO12:CHARACTER*6   ПЕРЕМЕННАЯ, СОДЕРЖАЩАЯ
C                                     КОЛОНКИ 7—12;
C    PAGENO:     ЧИСЛО, КОТОРОЕ ПЕЧАТАЕТСЯ В ВЕРХНЕЙ ЧАСТИ ТЕКУ-
C                ЩЕЙ СТРАНИЦЫ;
C    LINCNT:     СЧЕТЧИК СТРОК, ОСТАВШИХСЯ НА ТЕКУЩЕЙ СТРАНИЦЕ.
C
C  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:
C    OUTPUT:     ДЛЯ ФОРМИРОВАНИЯ И ПЕЧАТИ СЛЕДУЮЩЕЙ СТРОКИ;
C    HEADER:     ДЛЯ НАЧАЛА СЛЕДУЮЩЕЙ СТРАНИЦЫ И ВОССТАНОВЛЕ-
C                НИЯ СЧЕТЧИКА СТРОК.
C  ИНСТРУКЦИИ-ФУНКЦИИ:
C    LETTER(C):  ЛОГИЧЕСКАЯ — ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ «C» БУКВОЙ;
C    DIGIT(C):   ЛОГИЧЕСКАЯ — ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ «C» ЦИФРОЙ;
C    * * ЗАМЕЧАНИЕ * * ОБЕ ФУНКЦИИ ПРЕДПОЛАГАЮТ КОД EBCDIC
C                ИЛИ ASCII, ГДЕ НЕТ НИКАКИХ ЛИТЕР, КРОМЕ БУКВ
C                МЕЖДУ 'A' И 'Z', И НИКАКИХ ЛИТЕР, КРОМЕ ЦИФР
C                МЕЖДУ '0' И '9'.
C
C  ПРОГРАММИСТ: — — —                ДАТА: — — —
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * F
      INTEGER PAGENO, IRDR
      LOGICAL LETTER, DIGIT
      CHARACTER LINE*72, COL*1(72), NAM1*1(6), C*1
      CHARACTER*6 C1TO6, C7TO12, NAME
      EQUIVALENCE (LINE,COL(1),C1TO6), (COL(7),C7TO12), (NAME,NAM1(1))
      DATA IRDR/5/

        ОПРЕДЕЛЕНИЕ ИНСТРУКЦИЙ-ФУНКЦИЙ
      LETTER(C) = (C.GE.'A' .AND. C.LE.'Z')
      DIGIT(C) = (C.GE.'0' .AND. C.LE.'9')

      LINCNT = 0
      PAGENO = 1
```

Рис. 3.2. Программа, создающая PLM. (Текст в инстр. **100**: «Документ по логике работы программы», «Программа», «Стр:».)

```
C
C  *** ЦИКЛ ГЛАВНОЙ ПРОГРАММЫ ***
C
C        ВВЕСТИ КАРТУ, НЕ НАЧИНАЕТ ЛИ ОНА НОВЫЙ БЛОК КОММЕНТАРИЕВ
  100 READ(IRDR,101,END=999) LINE
  101 FORMAT(A72)
      IF (C1TO6 .NE. 'C * * ') GO TO 100
C
C        НАЙДЕНО НАЧАЛО БЛОКА КОММЕНТАРИЕВ
C        ПОИСК ИМЕНИ НОВОГО ПРОГРАММНОГО МОДУЛЯ
      NAME = ' '
      DO 200 I = 7,12
          IF (.NOT.LETTER(COL(I)) .AND. .NOT.DIGIT(COL(I))) GO TO 250
          NAM1(I-6) = COL(I)
  200 CONTINUE
C        ТЕПЕРЬ ЕСТЬ ИМЯ ИЗ 6 ЛИТЕР
  250 IF (COL(72) .EQ. 'M') NAME = 'M/PROG'
C
C        НАЧАТЬ НОВУЮ СТРАНИЦУ, ЕСЛИ ЭТО (A) НОВАЯ ГЛАВНАЯ ПРОГРАММА,
C        (B) ПОДПРОГРАММА ИЛИ (C) ТОЧКА ВХОДА И ЕСЛИ МЕНЕЕ 5 СТРОК
C        ОСТАЛОСЬ НА ТЕКУЩЕЙ СТРАНИЦЕ
      IF (COL(72).EQ.'M' .OR. COL(72).EQ.'S' .OR. LINCNT .LT.5)
     1    CALL HEADER (NAME, PAGENO, LINCNT)
C
C        ФОРМИРОВАНИЕ И ПЕЧАТЬ ОДИНОЧНОЙ СТРОКИ PLM
  275 CALL OUTPUT (COL, LINCNT, PAGENO, NAME)
C        ВЗЯТЬ НОВУЮ СТРОКУ И ПРОДОЛЖИТЬ ПЕЧАТЬ,
C        ПОКА НЕ ДОСТИГНЕМ КОНЦА БЛОКА КОММЕНТАРИЕВ
      READ (IRDR,101,END=999) LINE
      IF (C1TO6.EQ.'C * * ' .AND. COL(72).EQ.'F') GO TO 100
      GO TO 275
C
  999 STOP
      END
      SUBROUTINE OUTPUT (COL, LINCNT, PAGENO, NAME)


C  * * OUTPUT * * ФОРМИРОВАНИЕ И ПЕЧАТЬ СТРОК * * * * * * * * * S
C
C  ПОДПРОГРАММА ФОРМИРОВАНИЯ И ПЕЧАТИ ОДНОЙ СТРОКИ PLM
C
C  ВХОДНЫЕ ПАРАМЕТРЫ:
C    COL:        ПЕЧАТАЕМАЯ   СТРОКА — CHARACTER*1 — МАССИВ    ИЗ
C                72 ЭЛЕМЕНТОВ;
C    LINCNT:     СЧЕТЧИК СТРОК, ОСТАВШИХСЯ НА ТЕКУЩЕЙ СТРАНИЦЕ;
C    PAGENO:     НОМЕР ТЕКУЩЕЙ ВЫВОДНОЙ СТРАНИЦЫ;
C    NAME:       ИМЯ ТЕКУЩЕГО ПРОГРАММНОГО МОДУЛЯ.
C
C  ВЫХОДНЫЕ ПАРАМЕТРЫ:
C    LINCNT:     ИЗМЕНЕННЫЙ СЧЕТЧИК СТРОК
C                ПОСЛЕ ПЕЧАТИ ДАННОЙ СТРОКИ;
C    PAGENO:     УВЕЛИЧИВАЕТСЯ, ЕСЛИ НАЧАЛАСЬ НОВАЯ СТРАНИЦА.
C
C  ДЕЙСТВИЯ:     ЛИТЕРА 'C' В ПЕРВОЙ ПОЗИЦИИ СТРОКИ ЗАМЕЩАЕТСЯ
C                ПРОБЕЛОМ;
C                В КОЛОНКУ 72 ТАКЖЕ ЗАНОСИТСЯ ПРОБЕЛ
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * F


      INTEGER LINCNT, PAGENO, IPRT
      CHARACTER COL*1(72), NAME*6
      DATA IPRT/6/
C
C        ЕСТЬ ЛИ ЕЩЕ МЕСТО НА ТЕКУЩЕЙ СТРАНИЦЕ?
      IF (LINCNT .LT. 1) CALL HEADER(NAME, PAGENO, LINCNT)
C
```

Продолжение Рис 3.2.

```
      COL(1) = ' '
      COL(72) = ' '
      WRITE (IPRT,100) COL
100   FORMAT(1X,72A1)
      LINCNT = LINCNT-1
      RETURN
      END

      SUBROUTINE HEADER (NAME,PAGENO,LINCNT)
C * * HEADER * * ВЫВОД ЗАГОЛОВКА СТРАНИЦЫ * * * * * * * * * * * * * * * S
C
C  ФОРМИРОВАНИЕ И ПЕЧАТЬ ЗАГОЛОВКА СТРАНИЦЫ
C
C  ВХОДНЫЕ ПАРАМЕТРЫ:
C    NAME:   ИМЯ ТЕКУЩЕГО ПРОГРАММНОГО МОДУЛЯ
C    PAGENO: ЧИСЛО, НУМЕРУЮЩЕЕ СТРАНИЦУ
C
C  ВЫХОДНЫЕ ПАРАМЕТРЫ:
C    PAGENO: НОМЕР СЛЕДУЮЩЕЙ ВЫВОДНОЙ СТРАНИЦЫ
C            (ВХОДНОЕ ЗНАЧЕНИЕ PAGENO +1);
C    LINCNT: КОЛИЧЕСТВО СТРОК, КОТОРЫЕ МОГУТ БЫТЬ НАПЕЧАТАНЫ
C            НА НОВОЙ СТРАНИЦЕ
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * F
C
      INTEGER PAGENO, LINCNT, IPRT, MAXLN
      CHARACTER NAME*6
      DATA IPRT/6/, MAXLN/56/
      WRITE(IPRT,100) NAME, PAGENO
  100 FORMAT ('1', 2X, 'PROGRAM LOGIC MANUAL',  10X,
     1  'ROUTINE: ',  A6,    T61, 'PAGE: ', I4 / )
      PAGENO = PAGENO+1
      LINCNT = MAXLN
      RETURN
      END
```

Продолжение Рис 3.2.

```
ДОКУМЕНТ ПО ЛОГИКЕ РАБОТЫ ПРОГРАММЫ: ПРОГРАММА M/PROG СТР:1
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ПРОГРАММА ПОСТРОЕНИЯ PLM
(ДОКУМЕНТА ПО ЛОГИКЕ РАБОТЫ ПРОГРАММЫ)
        ПРОГРАММИСТ: — — —                ДАТА: — — —

   ПРОГРАММА ПРИНИМАЕТ ТЕКСТ ДРУГОЙ ПРОГРАММЫ НА ИСХОДНОМ
   ЯЗЫКЕ И ИЗВЛЕКАЕТ БЛОКИ КОММЕНТАРИЕВ ИЗ КАЖДОЙ ПОДПРО-
      ГРАММЫ ДЛЯ СОЗДАНИЯ PLM

   ЛОГИКА РАБОТЫ:
      ВВОДЯТСЯ КАРТЫ И ОСУЩЕСТВЛЯЕТСЯ ПОИСК ПОСЛЕДОВАТЕЛЬНО-
      СТИ C * * В КОЛОНКАХ 1—6. ЕСЛИ ТАКАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ
      НАЙДЕНА, ТО ЭТО ОТМЕЧАЕТ НАЧАЛО БЛОКА КОММЕНТАРИЕВ.
      АНАЛИЗИРУЕТСЯ КОЛОНКА 72 ДЛЯ ОПРЕДЕЛЕНИЯ:
                  ГЛАВНОЙ ПРОГРАММЫ (КОЛ. 72='M')
                  ПОДПРОГРАММЫ      (КОЛ. 72='S')
                  ТОЧКИ ВХОДА       (КОЛ. 72='E')
      ЕСЛИ ЭТО ГЛАВНАЯ ПРОГРАММА, ТО МОДУЛЬ БУДЕТ НАЗЫВАТЬСЯ
                  'M/PROG'.
      ЕСЛИ ЭТО ПОДПРОГРАММА ИЛИ ТОЧКА ВХОДА, ТО ИМЕНЕМ СТАНО-
                  ВИТСЯ ЦЕПОЧКА БУКВ, СЧИТАННАЯ В КОЛОНКАХ 7—12.
       ЭТО ИМЯ ПЕЧАТАЕТСЯ В ВЕРХНЕЙ ЧАСТИ КАЖДОЙ СТРАНИЦЫ PLM.
                  ЛИСТИНГ ИНСТРУКЦИЙ ПРОДОЛЖАЕТСЯ В PLM ДО ТЕХ
      ПОР, ПОКА НЕ БУДЕТ СЧИТАНА СЛЕДУЮЩАЯ КАРТА, СОДЕРЖАЩАЯ
      C' * *' В КОЛОНКАХ 1—6 И 'F' В КОЛОНКЕ 72.
      ЗАТЕМ ПРОДОЛЖАЕТСЯ ПОИСК ПЕРВОЙ ПЕРФОКАРТЫ БЛОКА КОМ-
                  МЕНТАРИЕВ НОВОГО ПРОГРАММНОГО МОДУЛЯ.
```

Рис. 3.3. Образец распечатки программы, создающей PLM.

```
ОСНОВНЫЕ ПЕРЕМЕННЫЕ:
  NAME: ИМЯ МОДУЛЯ (ОСНОВНОЙ ПРОГРАММЫ, ПОДПРОГРАММЫ ИЛИ
        ТОЧКИ ВХОДА);
  LINE: ТОЛЬКО ЧТО СЧИТАННАЯ КАРТА. ПОСРЕДСТВОМ ЭКВИВАЛЕНТ-
        НОСТИ ПОЭТОМУ МОЖНО ССЫЛАТЬСЯ НА:
        COL:CHARACTER*1          МАССИВ ИЗ 72 ЭЛЕМЕНТОВ;
        C1TO6:CHARACTER*6        ПЕРЕМЕННАЯ, СОДЕРЖАЩАЯ
                                 КОЛОНКИ 1—6;
        C7TO12:CHARACTER*6       ПЕРЕМЕННАЯ, СОДЕРЖАЩАЯ
                                 КОЛОНКИ 7—12;
  PAGENO:  ЧИСЛО, КОТОРОЕ ПЕЧАТАЕТСЯ В ВЕРХНЕЙ ЧАСТИ ТЕКУ-
           ЩЕЙ СТРАНИЦЫ;
  LINCNT:  СЧЕТЧИК СТРОК, ОСТАВШИХСЯ НА ТЕКУЩЕЙ СТРАНИЦЕ.
ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:
  OUTPUT:  ДЛЯ ФОРМИРОВАНИЯ И ПЕЧАТИ СЛЕДУЮЩЕЙ СТРОКИ;
  HEADER:  ДЛЯ НАЧАЛА СЛЕДУЮЩЕЙ СТРАНИЦЫ И ВОССТАНОВЛЕ-
           НИЯ СЧЕТЧИКА СТРОК.
ИНСТРУКЦИИ — ФУНКЦИИ:
  LETTER(C): ЛОГИЧЕСКАЯ — ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ «С» БУКВОЙ;
  DIGIT(C):  ЛОГИЧЕСКАЯ — ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ «С» ЦИФРОЙ;
  * * ЗАМЕЧАНИЕ * * ОБЕ ФУНКЦИИ ПРЕДПОЛАГАЮТ КОД EBCDIC
        ИЛИ ASCII, ГДЕ НЕТ НИКАКИХ ЛИТЕР, КРОМЕ БУКВ
        МЕЖДУ 'A' И 'Z'; И НИКАКИХ ЛИТЕР, КРОМЕ ЦИФР
        МЕЖДУ '0' И '9'.

РУКОВОДСТВО ПО ЛОГИКЕ РАБОТЫ ПРОГРАММЫ ПРОГРАММА: OUTPUT СТР:2

* * OUTPUT * * ФОРМИРОВАНИЕ И ПЕЧАТЬ СТРОК * * * * * *

ПОДПРОГРАММА ФОРМИРОВАНИЯ И ПЕЧАТИ ОДНОЙ СТРОКИ PLM

ВХОДНЫЕ ПАРАМЕТРЫ:
  COL:     ПЕЧАТАЕМАЯ СТРОКА — CHARACTER* 1 МАССИВ ИЗ
           72 ЭЛЕМЕНТОВ;
  LINCNT:  СЧЕТЧИК СТРОК, ОСТАВШИХСЯ НА ТЕКУЩЕЙ СТРАНИЦЕ;
  PAGENO   НОМЕР ТЕКУЩЕЙ ВЫВОДНОЙ СТРАНИЦЫ;
  NAME     ИМЯ ТЕКУЩЕГО ПРОГРАММНОГО МОДУЛЯ.

ВЫХОДНЫЕ ПАРАМЕТРЫ:
  LINCNT   ИЗМЕНЕННЫЙ СЧЕТЧИК СТРОК: СКОЛЬКО ОСТАЛОСЬ
           ПОСЛЕ ПЕЧАТИ ДАННОЙ СТРОКИ;
  PAGENO   УВЕЛИЧИВАЕТСЯ, ЕСЛИ НАЧАЛАСЬ НОВАЯ СТРАНИЦА

ДЕЙСТВИЯ:  ЛИТЕРА 'C' В ПЕРВОЙ ПОЗИЦИИ СТРОКИ ЗАМЕЩАЕТСЯ
           ПРОБЕЛОМ;
           В КОЛОНКУ 72 ТАКЖЕ ЗАНОСИТСЯ ПРОБЕЛ

РУКОВОДСТВО ПО ЛОГИКЕ РАБОТЫ ПРОГРАММЫ ПРОГРАММА: HEADER СТР:3

* * HEADER * * ВЫВОД ЗАГОЛОВКА СТРАНИЦЫ * * * * * * * * *

ФОРМИРОВАНИЕ И ПЕЧАТЬ ЗАГОЛОВКА СТРАНИЦЫ

ВХОДНЫЕ ПАРАМЕТРЫ:
  NAME:    ИМЯ ТЕКУЩЕГО ПРОГРАММНОГО МОДУЛЯ;
  PAGENO:  ЧИСЛО, НУМЕРУЮЩЕЕ СТРАНИЦУ.
ВЫХОДНЫЕ ПАРАМЕТРЫ:
  PAGENO:  НОМЕР СЛЕДУЮЩЕЙ ВЫВОДНОЙ СТРАНИЦЫ
           (ВХОДНОЕ ЗНАЧЕНИЕ PAGENO+1)
  LINCNT:  КОЛИЧЕСТВО СТРОК, КОТОРЫЕ МОГУТ БЫТЬ НАПЕЧА-
           ТАНЫ НА НОВОЙ СТРАНИЦЕ.
```

Продолжение Рис. 3.3.

программисты следуют этим рекомендациям. Мы рассмотрим программу, которая принимает на входе плохо оформленные программы на Фортране и в соответствии с изложенными рекомендациями производит в них косметические изменения.

При построении алгоритма мы, по существу, будем идти тем же путем, по которому прошел автор этой программы. Следует подчеркнуть, что это, конечно, не единственный из возможных способов разработки алгоритма. В книге демонстрируются

различные стили программирования и составления документации. Вам необходимо выработать свой собственный стиль, естественный и эффективный для решения ваших задач.

Итак, перед нами стоит задача: дана программа на Фортране, требуется построить идентичную программу, отличающуюся только внешним оформлением. Требуемые изменения сводятся к следующему:

1. Предварительно установить начало всех инструкций в колонке 7.
2. При появлении слова DO сместить начало каждой последующей строки на три позиции вправо по отношению к текущему началу.
3. При появлении слова CONTINUE сдвинуть начало последующих строк на три позиции влево.
4. Никогда не начинать строки далее колонки 25 (при глубине в шесть уровней) и раньше колонки 7 (без отступа).
5. Не изменять положения инструкций FORMAT.
6. Если нужно, по всей программе заменить вхождения определенной цепочки какой-либо другой цепочкой (например, изменить имя одной переменной на другое).
7. Полагаем, что в ключевых словах FORMAT, DO или CONTINUE нет пробелов. (Справа и слева от этих слов, конечно, могут быть пробелы.)

После того как мы в общих чертах сформулировали задачу, можно перейти к основным этапам ее решения:

1. Прочитать перфокарту и найти начало инструкции.
2. Проверить: нет ли ключевых слов DO, CONTINUE или FORMAT.
3. Найти образец, по которому требуется произвести изменения.
4. Вывести измененный текст, следя за строками продолжения.
5. Вернуться к шагу 1 для обработки всех инструкций программы.

Этот черновой набросок действий — в сущности первый этап нисходящего конструирования программы. Поскольку логика обработки проста, мы не приводим полную блок-схему или алгоритм ее реализации. При разработке программы, преобразующей формат, мы вновь встречаемся с тремя видами операций над литерами — пересылкой текста, поиском определенных групп литер и поиском первой литеры, отличной от пробела. Три перечисленных вида операций над литерами удобно выделять в подпрограммы. Поэтому мы составим и отладим подпрограммы

```
      SUBROUTINE CHMOVE (TO,TOPOS, TOLEN, FROM, FRPOS, FRLEN, MOVLEN)
C** CHMOVE  ********************************************** S
C
C  УНИВЕРСАЛЬНАЯ ПОДПРОГРАММА ПЕРЕСЫЛКИ ЛИТЕР
C
C  ПЕРЕСЫЛАЕТ ДАННЫЕ ИЗ МАССИВА „FROM“ В МАССИВ „TO“.
C  МАКСИМАЛЬНЫЕ РАЗМЕРЫ ЭТИХ МАССИВОВ „FRLEN“ И „TOLEN“.
C  ПОЗИЦИИ, ИЗ КОТОРЫХ И В КОТОРЫЕ ВЫПОЛНЯЕТСЯ ПЕРЕСЫЛКА, СОДЕРЖАТСЯ,
C    СООТВЕТСТВЕННО, В „FRPOS“ И „TOPOS“.
C  КОЛИЧЕСТВО ПЕРЕСЫЛАЕМЫХ ДАННЫХ УКАЗАНО В ПЕРЕМЕННОЙ „MOVLEN“.
C
C  РЕГИСТРИРУЕМЫЕ ОШИБКИ:
C  ЕСЛИ КОЛ-ВО ПЕРЕСЫЛАЕМЫХ ЛИТЕР ПРЕВЫШАЕТ РАЗМЕР МАССИВА „FROM“ ИЛИ „TO“,
C    ТО ПЕРЕСЫЛАЮТСЯ ТОЛЬКО ЛИТЕРЫ В ПРЕДЕЛАХ МАССИВА.
C
C  ПЕРЕСЫЛКА ВЫПОЛНЯЕТСЯ ПОЛИТЕРНО, СЛЕВА НАПРАВО.
C  МАССИВЫ „TO“ И „FROM“ МОГУТ, ЕСЛИ НУЖНО, ПЕРЕСЕКАТЬСЯ
C
C  ПРОГРАММИСТ: ---                          ДАТА: ---
C
C ********************************************** F

      INTEGER TOLEN, FRLEN, TOPOS, FRPOS, MOVLEN
      CHARACTER*1 TO(TOLEN), FROM(FRLEN)
C
C. . . . . .ОПРЕДЕЛИМ МАКС. КОЛИЧЕСТВО ПЕРЕСЫЛАЕМЫХ ЛИТЕР
C. . . . . .TOLEN-TOPOS+1 = МАКС. КОЛ-ВО ДАННЫХ, ОСТАВШИХСЯ В МАССИВЕ „TO“
C. . . . . .ТО ЖЕ САМОЕ ДЛЯ МАССИВА „FROM“
      MOVLEN = MINO (MOVLEN, TOLEN-TOPOS+1, FRLEN-FRLEN+1)
      IF (MOVLEN .LE. 0) RETURN
C
C. . . . . .ПРАВИЛЬНОСТЬ ПЕРЕСЫЛКИ ОБЕСПЕЧЕНА; ПО 1 ЛИТЕРЕ:
      DO 10 I = 1, MOVLEN
         TO(TOPOS+I-1) = FROM(FRPOS+I-1)
   10 CONTINUE
      RETURN
      END

      SUBROUTINE CHFIND (TARGET, PATTRN, PATL, COL, SKIP, FOUND)
C** CHFIND********************************************** S
C
C  ПРОГРАММА ИЩЕТ ЦЕПОЧКУ ЛИТЕР
C
C  ПОИСК ЦЕПОЧКИ „PATTRN“ ДЛИНОЙ „PATL“ В ЦЕПОЧКЕ „TARGET“.
C  ПОИСК НАЧИНАЕТСЯ В ПОЗИЦИИ „COL“ ЦЕПОЧКИ „TARGET“.
C
C  ЕСЛИ В „TARGET“ НЕ ОСТАЛОСЬ ПОЗИЦИЙ „PATL“,
C    Т.Е. COL+PATL-1 .GT. CARDL (=80), ТО
C    FOUND <=.FALSE. И ВОЗВРАТ
C
C  ЕСЛИ СЛЕДУЮЩИЕ ПОЗИЦИИ „PATL“ И ЦЕПОЧКИ „TARGET“ СООТВЕТСТВУЮТ
C    „PATTRN“, ТО „FOUND“<=.TRUE., COL НЕ ИЗМЕНЯЕТСЯ
C    И ВЫПОЛНЯЕТСЯ ВОЗВРАТ.
C  SKIP УКАЗЫВАЕТ, ДОЛЖНЫ ЛИ ЛИТЕРЫ СОВПАДАТЬ С ПОЗИЦИИ „COL“
C    ИЛИ СО СЛЕДУЮЩИХ ПОЗИЦИЙ.
C  ЕСЛИ „SKIP“ РАВНО.TRUE., ТО ПРИ НЕУДАЧЕ В ПОЗИЦИИ „COL“
C    ПОИСК ПРОДОЛЖАЕТСЯ В ПОЗИЦИЯХ COL+1, COL+2, ..., ПОКА
C    (А) НЕ БУДЕТ ПОЛУЧЕНО СООТВЕТСТВИЕ. ТОГДА:
C. . . . .COL <= КОЛОНКА, ГДЕ НАЧАЛОСЬ СОВПАДЕНИЕ
C. . . . .FOUND <= .TRUE. И ВОЗВРАТ
C    (В) „TARGET“ ИСЧЕРПАНА: СООТВЕТСТВИЯ НЕ НАЙДЕНО В
C. . . . .COL, COL+1, . . . , COL+K И COL+(K+1)+PATL-1 .GT. CARDL
C. . . . .ЗАТЕМ „FOUND“ <= .FALSE., „COL“ ОСТАЕТСЯ НЕИЗМЕННЫМ
C. . . . .И ВЫПОЛНЯЕТСЯ ВОЗВРАТ.
C
C  ПРОГРАММИСТ: ---                          ДАТА: ---
C
C********************************************** F
      INTEGER PATL, COL, CARDL/80/
      LOGICAL SKIP, FOUND
      CHARACTER*1 TARGET(80), PATTRN(PATL)
      FOUND = .FALSE.
```

Рис. 3.4. Программа, изменяющая формат карт.

```
C. . . . . .СНАЧАЛА ПРОВЕРИТЬ ПРАВИЛЬНОСТЬ ВХОДНЫХ ДАННЫХ
      IF (PATL .LT 1 .OR. COL .LT. 1) RETURN
C. . . . . .ПРОВЕРИТЬ, ЧТО ПОЗИЦИИ „PATL" НАХОДЯТСЯ В „TARGET"
      IF (COL+PATL-1 .GT. CARDL) RETURN
C. . . . . .НАЧАЛО ПОИСКА С ПОЗИЦИИ „COL"
      ISTART = COL
      IEND = COL
C. . . . . .ЕСЛИ SKIP РАЗРЕШЕН, ОПРЕДЕЛИМ ПОСЛЕДНЕЕ ВОЗМОЖНОЕ НАЧАЛО
      IF (SKIP) IEND = CARDL-PATL+1
C
C. . . . . .ЕСЛИ ПЕРВАЯ БУКВА ПОДХОДИТ, ПРОВЕРИТЬ ВСЮ ЦЕПОЧКУ
      DO 100 J = ISTART, IEND
         IF (TARGET(J) .NE. PATTRN(1)) GO TO 100
         IF (PATL .EQ. 1) GO TO 200
C. . . . . .СРАВНЕНИЕ НА ПОЛНОЕ СООТВЕТСТВИЕ
            DO 50 K = 2, PATL
               IF (TARGET(J+K-1) .NE. PATTRN(K)) GO TO 100
   50       CONTINUE
            GO TO 200
  100 CONTINUE
C. . . . . .ЕСЛИ ВЫШЛИ СЮДА, НЕ НАЙДЕНО - ВОЗВРАТ
      RETURN
C
C. . . . . .НАЙДЕНО ПРАВИЛЬНОЕ СООТВЕТСТВИЕ
  200 FOUND = .TRUE.
      COL = J
      RETURN
      END


      SUBROUTINE NOBLNK (LINE, COL, FOUND)
C** NOBLNK*********************************** S
C
C   ЭТА ПОДПРОГРАММА НАХОДИТ СЛЕДУЮЩУЮ ОТЛИЧНУЮ ОТ ПРОБЕЛА
C   ЛИТЕРУ В МАССИВЕ „LINE" РАЗМЕРА 80, НАЧИНАЯ С ПОЗИЦИИ „COL"
C
C   ПРИ ПОИСКЕ АНАЛИЗИРУЮТСЯ ПОЗИЦИИ COL, COL+1, COL+2,..., ПОКА:
C    (A) НЕ БУДЕТ НАЙДЕНА ОТЛИЧНАЯ ОТ ПРОБЕЛА ЛИТЕРА;
C. . . . . .ТОГДА COL ВОЗВРАЩАЕТСЯ С ПОЗИЦИЕЙ ПРОБЕЛА
C. . . . . .И FOUND = .TRUE., ИЛИ
C    (B) НЕ НАЙДЕНО НИ ОДНОЙ ОТЛИЧНОЙ ОТ ПРОБЕЛА ЛИТЕРЫ; ТОГДА
C. . . . . .COL НЕ ИЗМЕНЯЕТСЯ И
C. . . . . .FOUND = .FALSE.
C
C    ПРОГРАММИСТ: ---                          ДАТА: ---
C
C*********************************************** F


      INTEGER COL
      LOGICAL FOUND
      CHARACTER*1 BLANK/'*'/, LINE(80)
C. . . . . .ПОКА СООТВЕТСТВИЯ НЕТ; ИЗМЕНИМ, ЕСЛИ НАЙДЕМ СООТВЕТСТВИЕ
      FOUND = .FALSE.
      IF (COL .GT. 80 .OR. COL .LE. 0) RETURN
C
C. . . . . .ПРОВЕРЯЕМ КАЖДУЮ КОЛОНКУ
      DO 10 I=COL, 80
         IF (LINE(I) .NE. BLANK) GO TO 20
   10 CONTINUE
C. . . . . .ЕСЛИ ВЫШЛИ СЮДА, СООБЩАЕМ О НЕУДАЧЕ
      RETURN
C
C. . . . . .ЛИТЕРА НАЙДЕНА. СООБЩАЕМ ОБ УСПЕХЕ
   20 FOUND = .TRUE.
      COL = I
      RETURN
      END
C
C              ======================
C              == ГЛАВНАЯ ПРОГРАММА ==
C              ======================
C
```

Продолжение Рис. 3.4.

```
C*********************************************** M
C
C  РЕФОРМАТОР ПЕРФОКАРТ
C
C  ЭТА ПРОГРАММА ИЗМЕНЯЕТ ФОРМАТ ПРОГРАММ
C
C  ОНА ДЕЛАЕТ ОТСТУП В ЦИКЛАХ И ИЗМЕНЯЕТ ДРУГИЕ ЦЕПОЧКИ.
C  ПРОГРАММА ВВОДИТ ПЕРФОКАРТУ, РАЗБИТУЮ НА 4 КОМПОНЕНТЫ
C                          (STMTNO, CONT, INCAR, SEQFLD)
C  И ПРЕОБРАЗУЕТ ЭТУ КАРТУ
C
C  ОСНОВНЫЕ ПЕРЕМЕННЫЕ:
C  INCAR(66):  СЧИТЫВАЕМАЯ КАРТА
C  OUTCAR(132):  СЛЕДУЮЩАЯ КАРТА ВЫВОДА
C  INDENT:  ЧИСЛО ПРОБЕЛОВ ДЛЯ СЛЕДУЮЩЕЙ СТРОКИ
C  DELSTR:  УНИЧТОЖАЕМАЯ ЦЕПОЧКА ДЛИНОЙ „DELLEN“
C  REPL:  ЦЕПОЧКА ДЛИНОЙ „REPLEN“ ЗАМЕЩАЕТ УНИЧТОЖЕННУЮ
C
C  ПРОГАММИСТ: ---                          ДАТА: ---
C
C*********************************************** F
C


      EQUIVALENCE (STMTNO,COL1)
      LOGICAL ANCHOR, FOUND
      CHARACTER*1 INCAR(66),OUTCAR(132),COL1
      CHARACTER SEQFLD*8, STMTNO*5, CONT*1
      CHARACTER*1 FORMAT(6), DO(2), CONTIN(8), DELSTR(5), REPL(4)
      INTEGER DELLOC, INDENT, ENDLOC, NEXTIN, DELLEN,
     X   REPLEN, CARDLN, OUTUNT, COL, REPLOC, TOPOS
      LOGICAL ANCHOR, FOUND
      EQUIVALENCE (STMTNO, COL1)
      DATA FORMAT/'F','O','R','M','A','T'/, DO/'D','O'/
      DATA CONTIN/'C','O','N','T','I','N','U','E'/
      DATA INUNIT/5/, OUTUNT/7/, CARDLN/66/
C. . . . . .НАПРИМЕР, ЗАМЕНИТЬ „QWXYZ“ НА „XYZQ“
      DATA REPLEN/4/,REPL/'X','Y','Z','Q'/, ANCHOR/.TRUE./
      DATA DELLEN/5/,DELSTR/'Q','W','X','Y','Z'/, OUTCAR/132*' '/
      NEXTIN=1
   90 READ (INUNIT,100,END=999) STMTNO,CONT,INCAR,SEQFLD
  100 FORMAT (A5,A1,66A1,A8)
C
C. . . . . .НАЧАЛО СКАНИРОВАНИЯ НОВОЙ КАРТЫ
  110 INDENT = NEXTIN
      IF (COL1 .EQ. 'C') GO TO 120
      COL = 1
C. . . . . .ВЗЯТЬ ПЕРВЫЙ НЕ-ПРОБЕЛ; ИГНОРИРОВАТЬ ПУСТУЮ КАРТУ
      CALL NOBLNK (INCAR,COL, FOUND, CARDLN)
      IF (.NOT. FOUND) GO TO 90
C
C. . . . . .КАРТА ФОРМАТ? ЕСЛИ ДА, НЕ ПРЕОБРАЗУЕМ
      CALL CHFIND (INCAR, FORMAT,6,COL,ANCHOR,FOUND,CARDLN)
      IF (.NOT. FOUND) GO TO 200
  120    WRITE (OUTUNT,100) STMTNO,CONT,INCAR,SEQFLD
           READ (INUNIT,100,END=999) STMTNO, CONT,INCAR, SEQFLD
           IF ((CONT.EQ.' ' .OR. CONT.EQ.'0').AND. COL1.NE.'C') GO TO 110
         GO TO 120
C
C. . . . . .КАРТА CONTINUE? СОКРАЩАЕМ ОТСТУП
  200 CALL CHFIND (INCAR,CONTIN,8,COL,ANCHOR,FOUND, CARDLN)
      IF (.NOT. FOUND) GO TO 250
         NEXTIN = MAX0(1, INDENT-3)
         GO TO 300
C. . . . . .КАРТА ЦИКЛА DO? УВЕЛИЧИВАЕМ ОТСТУП НА 6 ПОЗИЦИЙ
  250 CALL CHFIND (INCAR,DO,2,COL,ANCHOR,FOUND, CARDLN)
      IF (.NOT. FOUND) GO TO 300
         NEXTIN = MIN0(INDENT+3,19)
C
C. . . . . .ПОИСК ЦЕПОЧЕК, ПОДЛЕЖАЩИХ ВЫЧЕРКИВАНИЮ/ЗАМЕЩЕНИЮ
  300 REPLOC = COL
      CALL CHFIND (INCAR,DELSTR,DELLEN,REPLOC,.NOT.ANCHOR,FOUND,CARDLN)
      IF (.NOT.FOUND) GO TO 400
C. . . . . .НАЙДЕНА ЦЕПОЧКА ДЛЯ ЗАМЕЩЕНИЯ
C. . . . . .ПЕРЕСЛАТЬ ВПЛОТЬ ДО НАЧАЛА ЗАМЕЩЕНИЯ
```

Продолжение Рис. 3.4.

```
      TOPOS = INDENT
      MOVELN = REPLOC-COL+1
      CALL CHMOVE (OUTCAR,TOPOS,2*CARDLN,
     X     INCAR,COL,CARDLN, MOVELN)
      COL = COL+MOVELN+DELLEN
C. . . . . .ПЕРЕСЛАТЬ ЗАМЕЩАЕМУЮ ЧАСТЬ
      CALL CHMOVE (OUTCAR,TOPOS,2*CARDLN,REPL,1,REPLEN,REPLEN)
      MOVELN = CARDLN-COL
      TOPOS = TOPOS+REPLEN
C. . . . . .ПЕРЕСЛАТЬ КОНЕЦ ИНСТРУКЦИИ
      CALL CHMOVE (OUTCAR,TOPOS,2*CARDLN,
     X     INCAR,COL,CARDLN,MOVELN)
      ENDLOC = COL+MOVELN-1
      GO TO 500
C
C. . . . . .ПРОСТЕЙШИЙ СЛУЧАЙ: ЗАМЕЩЕНИЙ НЕТ
  400 CALL CHMOVE (OUTCAR,INDENT,2*CARDLN,INCAR,COL,CARDLN,CARDLN-COL+1)
      ENDCOL = INDENT+CARDLN-COL+1
C
C. . . . . .КАРТА, СФОРМИРОВАННАЯ В „OUTCAR“ ; ВЫВОД
  500 WRITE (OUTUNT,100) STMTNO,CONT,(OUTCAR(I),I=1,CARDLN),SEQFLD
      IF (ENDCOL .LE. CARDLN) GO TO 550
C. . . . . .ДЛЯ ДЛИННЫХ ИНСТРУКЦИЙ НЕОБХОДИМО ПРОДОЛЖЕНИЕ
      CALL NOBLNK (OUTCAR,CARDLN+1,FOUND,2*CARDLN)
      IF (.NOT. FOUND) GO TO 550
      CALL CHMOVE (OUTCAR, INDENT,2*CARDLN,
     X     OUTCAR,CARDLN+1,2*CARDLN, CARDLN-INDENT)
      ENDCOL = ENDCOL-(CARDLN-INDENT+1)
C. . . . . .ЗАПОЛНИТЬ ПРОБЕЛАМИ ПОЛЕ МЕТКИ И КОЛОНКИ 73-80
      SEQFLD = ' '
      STMTNO = ' '
      CONT = 'X'
      GO TO 500
C
C. . . . . .ЗАПОЛНИТЬ ПРОБЕЛАМИ ОСТАТОК ПЕРФОКАРТЫ
  550 J = 2*CARDLN
      DO 560 I = 1,J
      OUTCAR(I) = ' '
  560 CONTINUE
C
C. . . . . .ВВОДИМ СЛЕДУЮЩУЮ ВХОДНУЮ КАРТУ
      READ (INUNIT,100,END=999) STMTNO,CONT,INCAR,SEQFLD
      IF (CONT .NE. ' '.AND. CONT .NE. '0') GO TO 300
      GO TO 110
  999 STOP
      END
```

Продолжение Рис. 3.4.

CHMOVE, CHFIND и NOBLNK. Имея три указанные подпрограммы, было легко построить главную программу. Полный текст программы приведен на рис. 3.4.

Упражнения

3.01. Напишите процедуру INTEGER FUNCTION NXTBLK с тремя параметрами: STRING, STRLEN и IPOS: STRING — массив, имеющий длину STRLEN. IPOS — целая величина между 1 и STRLEN. Эта функция возвращает номер позиции первого обнаруженного пробела, при анализе массива, начиная с элемента STRING(IPOS); она возвращает —1, если IPOS имеет отрицательное значение, нулевое значение или большее STRLEN, и 0, если ни один из элементов не является пробелом.

3.02. Напишите процедуру INTEGER FUNCTION NONBLK с теми же параметрами, что и в предыдущем примере. Функция возвращает позицию первой в STRING отличной от пробела литеры.

3.03. Напишите подпрограмму NEXTWD, имеющую три параметра: BUFFER, POSN и LEN. BUFFER — это массив размером 80, POSN — число между 1 и 80. Подпрограмма анализирует элементы BUFFER (POSN), BUFFER (POSN+1), . . ., отыскивая первую буву. Затем, начиная с позиции, в которой найдена буква, подпрограмма анализирует последующие позиции, отыскивая первый пробел, знак пунктуации или конец буфера. Подпрограмма возвращает параметр POSN, значением которого является первая найденная буква, и параметр LEN, в котором содержится длина слова (начиная с позиции, в которой была найдена первая буква).

3.04. Напишите систему из шести функций типа LOGICAL: CEQ, CNE, CLE, CLT, CGE и CGT — с параметрами STR1, LEN1, STR2 и LEN2. Подпрограммы должны возвращать .TRUE. , если цепочка STR1 совпадает с цепочкой STR2. Цепочки сравниваются последовательно (литера с литерой) слева направо (т. е. от меньшего индекса к большему). Число сравниваемых литер равно наименьшей длине из LEN1 и LEN2. Первая же несовпавшая пара литер определит, какая из сравниваемых цепочек «меньше», а какая «больше». Если все сравниваемые литеры совпали, цепочки равны между собой при равенстве длин цепочек. Если длины цепочек не равны, но все литеры совпадают, цепочка с наименьшей длиной считается «меньшей». (*Замечание.* Можно написать одну функцию с несколькими входами.)

3.05. Напишите программу, выполняющую частичное моделирование шахматной игры. Рассмотрите игру на доске 8×8. Ваша программа должна случайным образом выбирать три клетки для расстановки ферзя, слона и ладьи (это фигуры вашего партнера) и одну клетку, где должен стоять ваш конь. Напечатайте расположение фигур на доске, используя для изображения четырех фигур следующие обозначения: Ф (ферзь), С (слон), Л (ладья) и К (конь). Затем проанализируйте клетки, на которые ваш конь может пойти. Пометьте эти клетки:

1 — Если конь может туда пойти, но будет побит одной из фигур партнера на следующем ходе.
2 — Если конь может пойти и не будет побит ни одной из фигур партнера на следующем ходе.
3 — Если конь может этим ходом взять какую-либо фигуру партнера.

Выведите на печать изображение доски с расположением фигур. Вспомните, что ферзь может перемещаться в любых направлениях — по горизонтали, вертикали и диагонали. Слон может перемещаться в любых направлениях только по диагонали, а ладья — по горизонтали и вертикали. Конь ходит на две клетки по горизонтали и одну по вертикали или на две клетки по вертикали и одну по горизонтали.

3.06. Один из методов определения авторства анонимных литературных произведений состоит в следующем. Вначале подсчитываются все слова анализируемого документа и строится таблица относительной частоты появления (в процентах) каждого слова. Затем эти проценты сравниваются с процентами из таблиц, построенных по известным произведениям предполагаемых авторов. Напишите программу, которая вводит часть текста и выделяет все слова в этом тексте (игнорируя знаки пунктуации и используя один или несколько пробелов в качестве разделителей между словами). Ваша программа должна печатать абсолютную и относительную частоту появления каждого слова в тексте.

3.07. Выравненным по правому краю текстом называется текст, в котором правые и левые края строк выравнены. Напишите процедуру, которая вводит строку текста и вставляет достаточное количество дополнительных пробелов между словами, чтобы подогнать строку до определенной длины.

3.08. (Для тех, кто имеет возможность работать в интерактивном режиме). Одной из интересных демонстрационных игр является старинная детская игра

в отгадывание слов. Игра состоит в следующем. Задумывается какое-нибудь слово и на дисплей выводятся черточки, по одной на каждую букву этого слова. Участник игры угадывает буквы слова. Если он угадал какую-нибудь букву, она высвечивается в той позиции (в тех позициях), в которой она стоит в слове. В любое время участник может попытаться угадать все слово. Цель игры состоит в том, чтобы определить выбранное слово при наименьшем числе попыток. Напишите программу этой игры. (Кстати, эта игра, конечно, с более трудными словами приводит в восторг даже взрослых.)

Глава 4

ВВОД/ВЫВОД: МАГНИТНЫЕ ЛЕНТЫ И ДИСКИ

Оперативная память ЭВМ — это та среда, в которой запоминаются программа, ее данные, результаты вычислений, строки, подготовленные для форматной печати. К каждому элементу данных внутри оперативной памяти существует прямой доступ. В языке Фортран такой доступ выполняется путем ссылки на элемент по его имени. Запись и чтение данных выполняются с очень большой скоростью, обычно превышающей миллион слов в секунду.

Однако на большинстве ЭВМ размер оперативной памяти довольно ограничен. Как следствие, количество оперативной памяти, выделяемой для программы, может оказаться недостаточным для хранения всех данных, которые должны быть одновременно доступны программе. Рассмотрим, например, программу, которая должна читать, сортировать и затем печатать фамилии и почтовые адреса 15 000 человек. Если длина одной записи имени/адреса составляет 80 литер, то потребовалась бы память емкостью более миллиона байтов, чтобы запомнить данные для этой программы. Сравнительно немногие машины могли бы выделить так много оперативной памяти для отдельной программы. Поэтому необходимо добавить к оперативной памяти менее дорогую память для того, чтобы разрешить проблемы подобного сорта.

Второй недостаток оперативной памяти — короткое время жизни данных, которые в ней могут храниться. После того как выполнение программы закончилось, выделенная ей оперативная память должна быть отдана другим программам. Поэтому любые значения, которые остаются в этой памяти, будут испорчены. Вернемся вновь к задаче о почтовых адресах. Что, если бы мы захотели сохранить указанные 15 000 отсортированных образов перфокарт для повторных распечаток? Можно было бы вводить снова исходную колоду карт, но для этого потребовалось бы каждый раз выполнять сортировку. Вспомнив, что сортировка 15 000 элементов требует в среднем около 200 000 сравнений ($15\,000 * \log_2(15\,000)$), мы вынуждены отвергнуть такое решение. Можно было бы отперфорировать отсортированную колоду перфокарт, но для 15 000 карт понадобилось бы 7,5 коробки.

Чтобы решить проблемы, возникающие при использовании в качестве носителя данных только перфокарт и оперативной памяти, в большинстве ЭВМ предусматриваются другие устройства хранения информации. Обычно широко используют два класса таких устройств: устройства, предназначенные для обработки данных в строго фиксированном порядке (с *последовательным доступом*), и устройства, которые обеспечивают обработку в произвольном порядке (с *прямым доступом*). Вслед за устройствами, работающими с единичными записями, такими, как устройство ввода с перфокарт, АЦПУ и др., наиболее широко распространенным устройством с последовательным доступом служит магнитная лента. Устройства хранения данных на магнитных дисках составляют наиболее общеупотребительный класс устройств с прямым доступом. Мы обсудим организацию и использование магнитных лент в последующих трех разделах; дисковые устройства рассмотрены в разд. 4.4 и 4.5.

4.1. Физическая организация магнитных лент

4.1.1. Запись данных

Бобины магнитных лент подобны тем, что используют на бытовых магнитофонах. Стандартная бобина умещает магнитную ленту длиной 2400 футов (ок. 730 м). Ширина ленты обычно достаточна для параллельной записи 9 битов данных. При принятом методе записи намагниченность в одном направлении означает двоичную единицу, а в другом направлении — нуль. Обычно данные записаны на 7 или 9 параллельных дорожках (ленты называются соответственно 7-дорожечной и 9-дорожечной). В любом случае число битов достаточно для кодирования одиночной литеры.

К коду каждой записываемой литеры добавляется бит, называемый *битом четности*. Поэтому для записи 6-разрядных литер в коде BCD требуется 7-дорожечная лента, а для 8-разрядного кода EBCDIC — 9-дорожечная. Для записи 7-разрядных литер в ASCII необходимо только 8 дорожек, поэтому на 9-дорожечной магнитной ленте один разряд всегда установлен в 0.

Наиболее распространенное применение разряда четности — контроль на нечетность. Если число единичных битов в литере четное, то разряд четности устанавливается равным 1; в противном случае — 0. Использование этого способа гарантирует, что каждая записанная на ленту литера содержит нечетное число битов, равных единице.

При записи данных на ленту для каждой литеры вычисляется бит четности и затем приписывается к ее коду. При чтении данных происходит проверка на нечетность кода, т. е. выясня-

ется, не произошла ли ошибка. Такой контроль позволяет обнаружить ошибку в одном разряде, но нельзя обнаружить ошибки, произошедшие одновременно в двух разрядах.

Цепочка литер представлена на ленте записью из следующих друг за другом литер. Между записанными на ленту литерами, т. е. между первой закодированной литерой и второй, между второй и третьей и т. д., имеются, конечно, интервалы (чистые участки ленты). Интервал примерно одинаков для любых данных, записанных на одной бобине, но для различных лент интервалы могут быть разными. Плотность, с которой литеры упакованы на ленте, обычно составляет 6250, 1600, 800, 556 или 200 литер на дюйм. Эти плотности обычно обозначают как 6250, 1600, 800, 556 и 200 *bpi*, где *bpi* — это сокращение от *bits per inch* (бит на дюйм). Так как все биты одиночной литеры расположены на ленте параллельно, то «бит на дюйм» равнозначно «литер на дюйм».

Емкость магнитной ленты зависит от ее длины и плотности записи. Магнитная лента длиной 2400 футов с плотностью записи 1600 *bpi* может хранить более 45 миллионов литер данных; только половину от этого количества — при плотности записи 800 *bpi*. При выборе плотности записи приходится идти на компромисс, поскольку, чем выше плотность, тем больше вероятность записи с ошибками (особенно на старых лентах) и выше стоимость устройств записи.

При необходимости записи на ленту более одного блока данных (цепочки литер) между блоками оставляют *промежуток*, чтобы различить отдельные блоки. Промежуток представляет собой чистый участок ленты. Длина промежутка обычно находится в пределах между 1/2 и 3/4 дюйма. Наличие этих промежутков снижает полезную емкость лент.

Рассмотрим, что произойдет, если мы заполним магнитную ленту длиной 2400 футов 80-литерными образами перфокарт. При использовании плотности записи 1600 *bpi* каждый блок занимает только 1/20 дюйма. Предположим, что промежуток, разделяющий блоки, имеет длину 1/2 дюйма. Тогда эффективная плотность составит 80 литер на 11/20 дюйма, или, что то же, 146 литер на дюйм. Следовательно, используется только 9% ленты. Однако если записывать по 3200 литер в блоке, каждый из которых содержит 40 образов карт, то эффективность использования ленты составляет примерно 80% (рис. 4.1).

Метод записи более чем одного элемента данных на блок называют *блокированием*. Каждый отдельный элемент данных в блоке называется *логической записью*, а каждый блок — *физической записью*. Данные считаются несблокированными, если каждая физическая запись включает только одну логическую запись. В основном данные, записанные на ленту, сблокированы, так

как применение больших физических записей обеспечивает высокий процент эффективного использования ленты. Как мы увидим позднее, блокирование также уменьшает время, требуемое для чтения или записи данных.

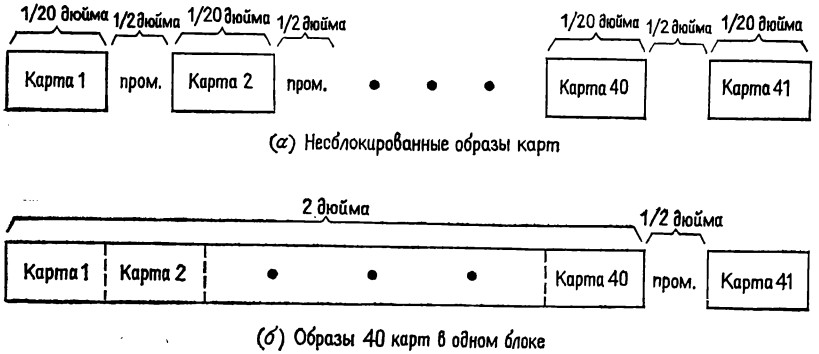


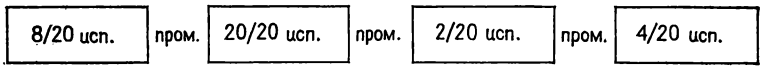
Рис. 4.1. Сблокированные и несблокированные записи.

Хотя блокирование данных имеет много достоинств, за них приходится платить. Для операций ввода или вывода внутри программы должна быть выделена достаточная область оперативной памяти, чтобы разместить как минимум одну физическую запись. (В большинстве случаев в памяти нужно хранить две или более физических записей.) Поэтому, если мы выбираем слишком большой размер блока, может не хватить памяти для выполнения программы. С другой стороны, если размер блока слишком мал, то бо́льшая часть ленты заполняется не данными, а промежутками.

К сожалению, нет магической формулы, которая диктовала бы оптимальный размер физической записи. Выбор должен основываться на таких факторах, как количество обрабатываемых данных и размер области памяти, отводимой для программы. Часто принимаются в расчет и другие моменты, однако это уже выходит за рамки данной книги. Мы будем использовать физические записи размерами в диапазоне от 600 до 4000 литер. Такой диапазон обычно приемлем для программ, выполняемых в небольшой области памяти и не требующих хранения больших объемов данных.

4.1.2. Записи фиксированной и переменной длины

На Фортране можно выводить в один и тот же файл на магнитной ленте записи различной длины. Изменением длины записей обычно управляют одним из трех методов.



(а) Несблокированные логические записи

(б) Логические записи в блоках по две

Рис. 4.2. Короткие записи, дополненные пробелами. Рисунок иллюстрирует, как записи небольшой длины записываются с использованием пустых литер. Предполагается, что максимальный размер логической записи равен 20. На ленту выводятся четыре записи — длиной 8, 20, 2 и 4.

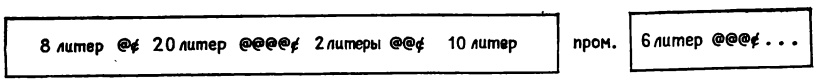
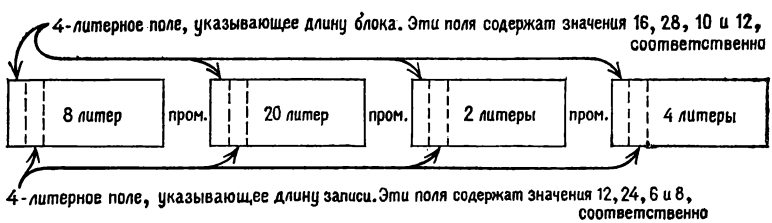
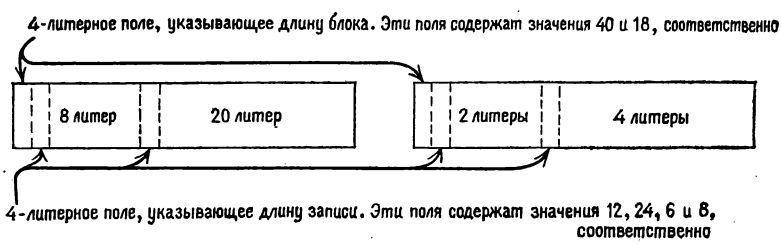


Рис. 4.3. Записи, в которых используются концевые маркеры. Рисунок показывает, как в системе DEC-10 хранятся записи с использованием маркера конца записи. Предполагается вывод четырех записей — длиной 8, 20, 2 и 16; каждая физическая запись имеет длину 10 слов (50 литер). Знак цента (¢) обозначает конец записи. Все записи дополнены до длины, кратной 5 (5 литер — это одно слово), посредством «пустых» литер (они обозначены через @).



(а) Несблокированные записи



(б) Сблокированные записи

Рис. 4.4. Записи переменной длины с использованием управляющих слов — система IBM. На рисунке показано, как хранятся записи переменной длины. Здесь изображены 4 записи длиной 8, 20, 2 и 4.

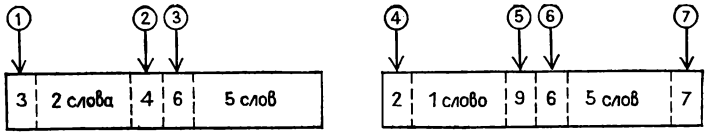


Рис. 4.5. Записи переменной длины с использованием управляющих слов — система DEC-10.

1. Стартовое слово первой записи. Указывает, что запись плюс стартовое слово занимают 3 слова.
2. Концевое слово первой записи. Указывает, что запись плюс стартовое и концевое слово занимают 4 слова.
3. Стартовое слово второй записи. Указывает, что первые 5 слов этой записи хранятся в данном блоке. Число 6 включает также стартовое слово.
4. Слово продолжения второй записи. Указывает, что далее следует оставшееся одно слово из записи номер 2. Число 2 включает также слово продолжения.
5. Концевое слово второй записи. Указывает, что в записи 9 слов, включая стартовое, концевое и слово продолжения.
6. Стартовое слово третьей записи. Указывает, что запись плюс стартовое слово занимают 6 слов.
7. Концевое слово третьей записи. Указывает, что запись плюс стартовое и концевое слова занимают 7 слов.

Рисунок показывает, как хранятся записи на ленте в системе DEC-10 с использованием управляющей информации. На ленте три записи — длиной 2, 6 и 5 слов. Длина каждой физической записи 10 слов.

Первое и самое простое решение состоит в том, чтобы всегда выводить логические записи некоторой фиксированной длины, которая равна наибольшей возможной длине записи. Данные занимают крайние левые позиции в логической записи, остаток заполняется «пустыми» литерами (например, пробелами). Этот метод проиллюстрирован на рис. 4.2.

Второе решение предполагает, что за каждой записью следует распознаваемый символ конца записи. При использовании этого метода можно упаковывать более одной логической записи в одном блоке. Можно даже разместить одну логическую запись в нескольких блоках, поскольку наличие маркера конца записи позволяет обнаруживать конец логической записи независимо от границ физической записи. Рис. 4.3 иллюстрирует, как концевые маркеры используются в системе DEC-10.

Третье решение — выводить записи, которым предшествуют некоторые управляющие литеры, определяющие фактическую длину данных. Системы ОС IBM 360-370 и DEC-10 используют этот метод для определенных типов выводных записей. Рис. 4.4 иллюстрирует метод применения записей переменной длины в машинах фирмы IBM. На рис. 4.5 показано, как этот метод реализован в системе DEC-10.

4.1.3. Проблема обновления записей

Если мы используем магнитную ленту для хранения данных, по которым, например, начисляется зарплата служащих, то может возникнуть необходимость заменить данные, хранящиеся в каком-то блоке, новой информацией. Если новый блок имеет ту же самую длину, что и старый, можно записать его непосредственно на место заменяемого блока. Такой процесс, который обычно называют *обновлением по месту*, не всегда может осуществляться успешно.

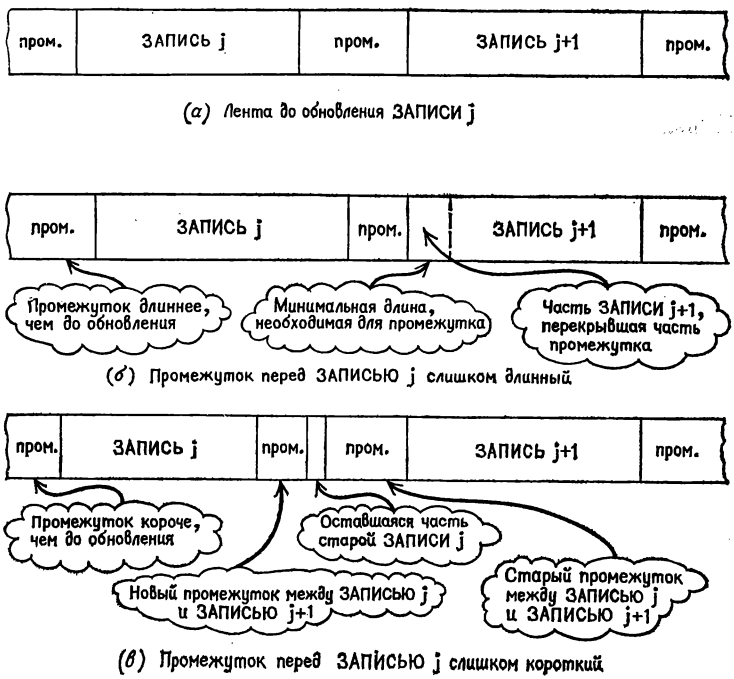


Рис. 4.6. Обновление записей на ленте.

Существо проблемы, связанной с переписью блока, состоит в том, что записывающие головки не могут быть установлены в точности так же, как при записи старого блока. В результате этой неточности концевая часть старого блока окажется нестертой или получится слишком малый промежуток между новым и следующим за ним старым блоком (рис. 4.6). Вследствие этого после переписи блока все последующие блоки оказываются недоступными для чтения.

4.1.4. Файлы данных

Группу связанных по смыслу записей, хранящихся на магнитной ленте, называют *файлом* (или в некоторых случаях *набором данных*). Одна бобина может содержать более одного файла. Действительно, магнитная лента имеет такую большую емкость, что обычно на нее записывают несколько файлов. Каждый файл на ленте оканчивается специальным маркером, называемым *конец файла*. За последним файлом на ленте следует маркер *конца тома* (в некоторых системах он представляется двумя последовательными маркерами конца файла).

На многих установках файлы можно помечать метками. Это означает, что непосредственно перед (а в некоторых случаях непосредственно за) каждым файлом располагается набор записей, называемых в совокупности *меткой*. Информация, записанная в метке, неодинакова в различных системах, но обычно включает: выбранное пользователем имя файла, которое должно быть известно тому, кто хочет прочитать файл; длину физических записей, содержащихся внутри файла; длину логических записей, из которых состоит каждая физическая запись.

Методы, посредством которых программа может определить, какой файл многофайловой ленты подлежит обработке, рассмотрены в разд. 4.3. Эти методы не являются частью Фортрана и специфичны для конкретной вычислительной системы.

4.1.5. Физические характеристики магнитной ленты

Устройства, которые читают данные с магнитных лент или записывают их на ленту, называются *лентопротяжными устройствами* (или магнитофонами, или лентопротяжками). Эти устройства имеют две оси для установки бобин с магнитной лентой. На одну ось устанавливают бобину с лентой, на другую — приемную пустую бобину. При обработке лента с первой бобины перематывается под головкой чтения/записи на приемную бобину.

Когда на устройство приходит запрос на чтение или запись, магнитная лента начинает двигаться мимо головки чтения / записи. Чтение или запись не начинается, пока лента не наберет полной скорости движения, которая обычно составляет от 75 до 125 дюймов в секунду. Такая скорость достигается после того, как лента продвинулась примерно на половину длины промежутка.

Если выполняется операция чтения, перед головкой чтения / записи должна находиться середина промежутка, предшествующего считываемому блоку данных. Когда начинается движение, головка чтения/записи следит за содержимым перематываемой магнитной ленты, ожидая начала данных. Как только

первая литера, расположенная за промежутком, будет обнаружена, начинается чтение. Чтение продолжается до тех пор, пока не будет обнаружено начало следующего промежутка. Лентопротяжное устройство постепенно замедляет ход, останавливаясь примерно посредине этого промежутка.

При выполнении операции записи головка чтения/записи должна находиться примерно в середине промежутка, расположенного после предшествующего блока. Головка чтения/записи стирает участок ленты во время пускового периода. Таким путем образуется необходимый промежуток перед новым блоком. Когда лента набирает полную скорость, записывается блок данных. После того как записана последняя литера, лента постепенно замедляет ход до полного останова. Во время этого периода головка чтения/записи стирает соответствующий участок, тем самым обеспечивая необходимый промежуток, отделяющий только что записанный блок от любой информации, которая может за ним последовать.

При выполнении операций на устройствах магнитной ленты возникают две проблемы. Первая из них связана с обновлением блоков на месте; она была рассмотрена в разд. 4.1.3. Вторая проблема касается длины блоков. При выполнении операции чтения предполагается, что считываемая головкой информация должна иметь некоторую минимальную длину, чтобы ее можно было интерпретировать как блок данных; в противном случае участок интерпретируется как «шум», вызванный работой лентопротяжного механизма или ошибкой записи. Минимальная длина физической записи отлична для разных устройств на магнитных лентах. Опыт, однако, показывает, что никогда не следует писать блоки длиной менее 20 литер.

4.1.6. Скорость передачи данных на магнитной ленте

К данным, находящимся в оперативной памяти ЭВМ, может быть обеспечен доступ с очень большой скоростью; для современных больших ЭВМ эта скорость составляет более 4 миллионов литер в секунду. Скорость передачи данных на ленту и с ленты значительно меньше. Рассмотрим ленту с плотностью записи 800 *bpi* и устройство, которое перемещает ленту со скоростью 75 дюймов в секунду. Если пренебречь временем начала движения и останова ленты, то можно считать, что данные передаются на ленту или с ленты со скоростью 60.000 литер в секунду.

Скорость передачи данных на ленте, хотя и меньше, чем в оперативной памяти, но значительно больше, чем скорость ввода данных с перфокарт. (Типичная скорость ввода с перфокарт — 1000 карт в минуту, или 1333 литеры в секунду.)

4.2. Инструкции Фортрана для работы с магнитной лентой

Каждая обрабатываемая программой на Фортране магнитная лента связана с некоторым номером канала. Выбор доступных номеров каналов и способы, посредством которых данная лента ассоциируется с каким-либо номером канала, коренным образом различаются для различных ЭВМ. В разд. 4.3 этот вопрос рассмотрен более детально. Здесь мы будем использовать имена вида TAPEx (где x — целое число) при каждом обращении к устройству на магнитной ленте. Соответствующее значение для указанных переменных должно быть задано до того, как представленные здесь сегменты программы могут быть скомпилированы.

Для работы с магнитными лентами в языке Фортран предусмотрено семь инструкций. К ним относятся: форматные инструкции READ и WRITE, бесформатные READ и WRITE, инструкции BACKSPACE, ENDFILE и REWIND.

4.2.1. Форматный ввод/вывод на магнитной ленте

Форматные инструкции READ и WRITE для работы с магнитными лентами записывают точно так же, как аналогичные инструкции ввода с перфокарт и вывода на печать; эти инструкции отличаются только номерами каналов.

Выполнение инструкций

```
      WRITE (TAPE1, 4000) (ALPHA(I), I=1, 100)
 4000 FORMAT (100A1)
```

обеспечивает создание 100-литерной логической записи и вывод ее на магнитную ленту устройства, обозначенного через TAPE1. Если записи не блокируются, данная отдельная логическая запись выводится как одна физическая запись. Если выполняется блокирование, то система обеспечивает слияние данной логической записи с другими записями, выводимыми на магнитную ленту. Операции по блокированию обычно скрыты от программиста.

Логические записи файла на магнитной ленте могут быть считаны только в том порядке, в котором они были записаны на ленту. Поэтому, если, например, мы хотим прочитать пятую логическую запись, необходимо вначале считать четыре предыдущие записи. Предполагая, что головка чтения/записи установлена точно перед записью номер 1, можно добиться желаемого результата посредством инструкций

```
      READ (TAPE1, 3000, END=999) (ALPHA(I), I=1,50)
 3000 FORMAT ////, 50A1)
```

Первые четыре слэша в формате позволяют прочесть и проигнорировать первые четыре записи. Поскольку список в формате полностью еще не исчерпан, считывается пятая запись. Если пятая запись содержит 50 или более литер, то первые 50 литер будут присвоены первым 50 элементам массива ALPHA, а любые оставшиеся в записи литеры игнорируются. Если запись короче 50 литер, будет выдано сообщение об ошибке точно так же, как если бы мы попытались ввести 100 литер с 80-колонной перфокарты. В некоторых ЭВМ это приводит к «фатальной» ошибке; в других происходит считывание следующей записи и значения, взятые из нее, используются для завершения операции чтения.

Предыдущая инструкция READ содержит необязательный параметр END=. Его применяют точно так же, как и в инструкции чтения перфокарт для обнаружения конца данных. В данном случае, если маркер конца файла будет обнаружен до пятой записи, то произойдет переход к инструкции с меткой 999.

4.2.2. Бесформатный ввод/вывод на магнитной ленте

При выполнении форматной инструкции WRITE значения данных, выдаваемых их оперативной памяти, преобразуются в соответствии с указанными форматными кодами. Часто форматная версия вещественных, целых или логических переменных требует больше литер, чем располагается в одном слове (длина внутреннего представления переменной). Например, если переменная записывает без форматного преобразования, она занимает четыре литеры в системе IBM 360-370 и пять литер в DEC-10. Однако если значение такой переменной записано с использованием форматного кода I9, то переменная будет представлена девятью литерами. Потребность в лишнем пространстве наряду с затратами времени и потерей точности, связанными с форматным преобразованием, может быть преодолена путем применения бесформатных инструкций чтения и записи.

Для того чтобы упаковать данные, записываемые на магнитную ленту, можно использовать бесформатную инструкцию WRITE. Общая форма инструкции:

WRITE *(номер канала) список вывода*

Отсутствие *номера формата* определяет, что инструкция бесформатная.

Выполнение инструкции

```
WRITE (TAPE1) (LIST(I), I=1,200)
```

приводит к записи на ленту без форматного преобразования 200 слов, содержащих первые 200 элементов массива LIST.

На машинах серий IBM 360-370 логическая запись для приведенной инструкции содержит 800 литер. На ЭВМ DEC-10 запись займет 1000 литер. В противоположность этому логическая запись, созданная инструкциями

```
      WRITE (TAPE1, 3000) (LIST(I), I=1,200)
 3000 FORMAT (200I9)
```

имеет длину 1800 литер.

Логические записи, которые были выведены на ленту с применением бесформатной инструкции WRITE, следует читать посредством бесформатной инструкции READ. Общая форма этой инструкции:

READ (*номер канала* [,END=*метка*]) [*список ввода*]

Список ввода определен в инструкции как необязательный. Его можно опустить, если желательно пропустить запись. Поэтому инструкция

```
READ (TAPE1)
```

пропускает одну логическую запись на ленте, ассоциированной с каналом TAPE1.

Хотя использование бесформатного ввода/вывода обычно считают выгодным, это не всегда так. Например, предположим, что мы хотим вывести на ленту логическую запись, содержащую значения 500 целых чисел — элементов массива A, и мы уверены, что все элементы имеют значения между 0 и 99. Форматный вывод

```
      WRITE (TAPE1, 4000) A
 4000 FORMAT (500I2)
```

обеспечит вывод на ленту записи длиной 1000 литер. Бесформатный вывод

```
WRITE (TAPE1) A
```

также обеспечивает вывод на ленту, но здесь запись будет содержать 2000 литер для машин серий IBM 360-370 и 2500 литер для ЭВМ DEC-10. Наверное, в этом случае лучше выбрать форматный вывод.

Форматный вывод необходим, когда лента должна читаться на машинах различных типов. Это связано с тем, что при бесформатном выводе данные представлены во внутреннем коде машины, который распознают только данная и подобные ей ЭВМ. Однако если мы используем форматный вывод, то магнитная лента может быть прочитана на двух ЭВМ с тем же самым литерным кодом (например, ASCII, BCD или EBCDIC).

4.2.3. Инструкция BACKSPACE

Инструкцию BACKSPACE, по-видимому, можно представлять себе как инструкцию реверса лентопротяжного механизма, чтобы обрабатывать только что записанную информацию. Поэтому, если выполняется инструкция вида

BACKSPACE *номер канала*

то следующей обрабатываемой логической записью будет запись, предшествующая той, которая была бы доступна в данный момент в отсутствие этой инструкции. Поэтому, например, первое чтение ленты

```
      READ (TAPE1,3000) (A(I), I=1,30)
3000  FORMAT (30I3)
      BACKSPACE TAPE1
      READ (TAPE1,3000) (B(I), I=1,30)
```

вызывает чтение логической записи, и первые 90 литер этой записи обрабатываются как 30 последовательных полей в формате I3. Значения указанных полей размещаются в первых 30 элементах массива A. Команда BACKSPACE указывает, что при следующем чтении должна обрабатываться та же самая логическая запись, которая только что была считана. Поэтому первым 30 элементам массива B будут присвоены те же самые значения, которые были присвоены элементам массива A.

Инструкция BACKSPACE не выполняет никаких действий в случае, когда лента установлена на чтение первой записи файла. Ясно, что нет смысла задавать обратное движение ленты от этой точки.

4.2.4. Инструкция ENDFILE

После того как в файл на магнитной ленте записаны все записи, должен быть записан маркер конца файла. В Фортране для этого используют инструкцию

ENDFILE *номер канала*

4.2.5. Инструкция REWIND

Общая форма инструкции REWIND (перемотать):

REWIND *номер канала*

Эта инструкция выполняется после окончания всей обработки файла. Кроме того, REWIND следует задавать всегда, когда необходимо вновь установить головку чтения/записи на логическое начало ленты. (Определение логического начала ленты раз-

личается для разных вычислительных систем. На машинах IBM 360-370 это означает начало обрабатываемого файла. На DEC-10 — начало первого файла на ленте.)

Магнитные ленты обычно используются для длительного хранения программ и данных. На рис. 4.7 приведена программа,

```
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* КОПИРОВАНИЕ КОЛОДЫ ПЕРФОКАРТ НА ЛЕНТУ.                                *
C* ЧТЕНИЕ И ПЕЧАТЬ СОДЕРЖИМОГО ЛЕНТЫ ДЛЯ КОНТРОЛЯ                        *
C* ПРОГРАММИСТ: ---                          ДАТА: ---                   *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER RDR/5/, PRT/6/, TAPE1/16/
      CHARACTER*80 CARD
C
C***        ЧТЕНИЕ ПЕРФОКАРТ И ЗАПИСЬ НА ЛЕНТУ
C
10    READ (RDR,15,END=20) CARD
15    FORMAT (A80)
      WRITE (TAPE1,15) CARD
      GO TO 10
20    ENDFILE TAPE1
C
C***        ЧТЕНИЕ ФАЙЛА НА ЛЕНТЕ И ВЫВОД СОДЕРЖИМОГО НА ПЕЧАТЬ
C
      REWIND TAPE1
      WRITE (PRT,25)
25    FORMAT ('1', T40, 'CONTENTS OF TAPE FILE' /
     1        '+', T40, '_______ __ ____ ____' ///)
30    READ (TAPE1,15,END=40) CARD
      WRITE (PRT,35) CARD
65    FORMAT (T11, A80)
      GO TO 30
40    REWIND TAPE1
      STOP
      END
```

Рис. 4.7. Программа, копирующая карты на ленту. (Текст в инстр. 25: «Содержимое файла на ленте.»)

которая копирует входную колоду перфокарт на ленту, перематывает ленту и выводит на печать ленточный файл для документирования его содержания. В приведенном примере опущены детали того, как ассоциируется номер канала в фортранной программе с действительным устройством на магнитной ленте. Следующий раздел дает некоторое представление о том, как это делается в системах OS IBM 360-370 и DEC-10.

4.3. Работа с магнитными лентами: системы IBM и DEC

Предположим, что мы правильно написали программу, обрабатывающую файл на магнитной ленте. Чтобы выполнить эту программу, необходимо знать соглашения, принятые на вашей ЭВМ. Эти соглашения могут существенно отличаться для различных машин и обычно незначительно отличаются для машин, использующих одну и ту же операционную систему.

В следующих двух подразделах рассмотрены примеры того, как номера каналов в программах на Фортране могут быть ассоциированы с файлами на магнитных лентах. Эти примеры относятся к системам IBM 360-370 и DEC-10.

4.3.1. Назначение устройствам в системе IBM OS/JCL

На машинах IBM серий 360-370 для управления операциями используются два принципиально различных языка.

Это OS/JCL и DOS/JCL (OS — Operating System, операционная система; DOS — дисковая операционная система; JCL — Job Control Language, язык управления заданиями).

OS/JCL — это язык управления, используемый в основном на больших машинах IBM 360-370. В этом языке устройствам назначают номера посредством команд, называемых командами определения данных (DD-команды). Поскольку некоторые аспекты спецификаций в DD-командах зависят от соглашений, принятых в конкретном вычислительном центре, мы здесь опустим много вопросов (например, расположение DD-команд в колоде перфокарт). Ограничимся лишь той информацией, которая применима в большинстве случаев, и отметим места, где вступают в силу локальные соглашения.

DD-команды, связанные с выполнением программ на Фортране, имеют следующую форму:

//[*имя-шага.*]FT*нк*F001 DD *операнды*

Две наклонные черты // должны быть пробиты в 1-й и во 2-й колонках перфокарты; их используют для указания того, что данная перфокарта содержит команду Языка управления заданиями (JCL). Номер канала в программе на Фортране, к которому мы хотим подключить физическое устройство, обозначен через *нк*.

Цепочка литер *имя-шага* определяет шаг задания; соглашения о ее форме различны для различных ВЦ, а иногда она может быть даже опущена. Вам следует узнать, нужно или нет задавать имя шага, и если нужно, то как.

Поле *операнды* представляет собой список из одного или нескольких значений операндов. Два обычно используемых операнда:

```
//FT11F001 DD      *
```

и

```
//FT09F001 DD SYSOUT=A
```

Первый операнд (*) указывает, что номер канала (в данном случае 11) означает канал ввода с перфокарт. Перфокарты должны следовать непосредственно за указанной DD-командой.

Сигналом для завершения ввода перфокарт из канала номер 11 служит или отсутствие дальнейших перфокарт в данном задании, или следующая управляющая перфокарта.

Во втором примере приведен операнд SYSOUT=A. Этот операнд объявляет, что все инструкции WRITE с данным номером канала (здесь 9) осуществляют вывод на АЦПУ.

DD-команды, аналогичные только что описанным, с номерами каналов 5 (*) и 6 (SYSOUT=A) обычно входят в ту стандартную часть языка управления заданиями, которая используется для программ на Фортране.

Для ввода с перфокарт редко применяются номера каналов, отличные от 5. Однако такая возможность может быть полезной, если программа будет выполняться на других вычислительных машинах. Например, стандартный номер канала для карточного файла в Фортране DEC-10 не 5, а 2. Для выполнения программы следует вставить перед перфокартами с данными DD-команду

```
//FT02F001 DD *
```

Это позволит ассоциировать с входным карточным файлом канал 2, а не 5. Все инструкции ввода с перфокарт следует задать в программе в форме READ (2, —) —.

Наиболее общепринятое употребление операнда SYSOUT=A— разрешить программам иметь доступ к двум или более файлам печати. Это средство дает нам чрезвычайно гибкие дополнительные возможности для выполнения отладки.

Предположим, что мы написали программу, которая направляет все отладочные сообщения в канал DBG. Тогда инструкции отладочного вывода WRITE имеют форму

WRITE (DBG, *формат*) *список*

Теперь, если мы захотим объединить отладочный листинг с обычным стандартным выводом, нужно инициализировать DBG посредством инструкции

```
DATA DBG/6/
```

Однако, если требуется, чтобы отладочный листинг был оформлен как отдельная группа страниц, независимая от вывода в канал с номером 6, можно включить, например, инструкцию

```
DATA DBG/9/
```

и DD-команду.

```
//FT09F001 DD SYSOUT=A
```

При выполнении программы будет создано два или более файлов печати, каждый из которых представляет собой обособ-

ленную часть программного вывода. Поэтому, если программа содержала последовательность инструкций

```
WRITE(6, —)   Список 1
WRITE(9, —)   Список 2
WRITE(6, —)   Список 3
```

где 6 и 9 — номера устройств печати, то мы не увидим строк, связанных со списком 2, между строками, относящимися к *списку 1* и *списку 3*. Вместо этого *список 3* будет следовать за *списком 1* в одной группе страниц, а *список 2* — в другой группе страниц, печатаемых устройством, которое подключено к каналу с номером 9.

Эффект, описанный в предыдущем абзаце, достигается за счет того, что все инструкции вывода WRITE с номером канала 6 направляют вывод в один файл, а все инструкции с номером канала 9 — в другой файл. На самом деле распечатка этих файлов выполняется после того, как выполнение программы закончилось. Следует сказать, что *буферизация вывода* (спулинг) применяется во многих современных вычислительных системах.

Значения операндов DD-команд для файлов на магнитной ленте гораздо более сложные, чем те, что были приведены для файлов на перфокартах и файлов печати. Мы познакомимся с этими операндами на ряде примеров, а затем кратко опишем назначение каждого используемого операнда. Здесь не делается попытки дать исчерпывающего изложения этого вопроса. Приведенные здесь сведения помогут лишь ознакомить вас с основными принципами. Более детальную информацию о DD-командах можно найти в соответствующих руководствах фирмы IBM и документации, имеющейся в вашем вычислительном центре.

Пример 1. Предположим, что мы хотим подключить к каналу 12 файл, имеющий следующие характеристики:

1. Файл должен быть записан на 9-дорожечную магнитную ленту [UNIT=TAPE].
2. Магнитная лента идентифицирована именем HUGHES [VOL=SER=HUGHES].
3. Файл должен быть создан при выполнении данного задания [DISP=NEW].
4. Файл помечен и должен быть первым файлом, записанным на данную магнитную ленту [LABEL=1].
5. Файл должен иметь имя MYDATA [DSN=MYDATA].
6. Файл записывается посредством форматной инструкции WRITE. Каждая запись имеет длину не более 100 литер. Все логические записи, выводимые на ленту, должны быть фиксированной длины по 100 литер. Каждый физический блок должен содержать 20 логических записей [DCB=

(RECEM=FB,LRECL=100,BLKSIZE=2000)]. Тогда необходимая DD-команда будет иметь вид

```
//FT12F001 DD UNIT=TAPE,VOL=SER=HUGHES,DISP=NEW,LABEL=1,
//            DSN=MYDATA,DCB=(RECFM=FB,LRECL=100,BLKSIZE=2000)
```

Заметим, что, поскольку управляющие перфокарты (JCL-карты) могут содержать информацию только до 71 колонки (а не до 72, как карты Фортрана), необходимы две такие перфокарты. Чтобы продолжить DD-команду, просто прерывают ее на запятой, следующей за каким-либо операндом, пробивают код // в колонках 1 и 2 следующей перфоркарты, пропускают как минимум 3 пробела и затем набивают оставшиеся операнды, начиная не дальше, чем с 16 колонки.

Пример 2. Метод, использованный в примере 1, требует, чтобы логические записи были всегда длиной 100 литер. Если же в записи содержится меньше данных, то она дополняется «пустыми» литерами (пробелами). Поэтому мы попытаемся переделать этот пример таким образом, чтобы записи выводились без дополняющих пустых литер. Для этого необходимо, чтобы блоки были переменной длины, как показано на рис. 4.4. В операндах DD-команды изменяем только DCB. Операнд DCB должен быть записан как

DCB=(RECEM=VB,LRECL=104,BLKSIZE=2084)]

В длине 104 учитывается 4-литерное поле, за которым следует поле для данных длиной до 100 литер. Длина 2084 включает 20*104 литер плюс еще одно 4-литерное поле.

Пример 3. Предположим, что необходимо подключить к каналу 8 файл на магнитной ленте, описанный следующим образом:

1. Файл записан на 9-дорожечной магнитной ленте [UNIT==TAPE].
2. Магнитная лента идентифицирована именем CPPO1 [VOL=SER=CPPO1].
3. Файл уже был создан [DISP=OLD].
4. Файл помечен и записан на ленте третьим файлом [LABEL==3].
5. Файл имеет имя YOURDATA [DSN=YOURDATA].
6. Логические записи содержат 350 литер, не сблокированы и должны быть считаны форматной инструкцией READ

[DCB=(RECEM=F, LRECL=350,BLKSIZE=350)].

Необходимая DD-команда имеет вид

```
//FT08F001 DD UNIT=TAPE,VOL=SER=CPPO1,DISP=OLD,LABEL=3,
//            DSN=YOURDATA,DCB=(RECFM=F,LRECL=350,BLKSIZE=350)
```

или то же самое без операнда DCB. Операнд DCB можно опустить, так как файл YOURDATA уже существует, и поэтому файл имеет метку, которая содержит значения DCB.

Пример 4. Файл должен быть записан на устройство 13. Он характеризуется следующим образом:

1. Файл должен быть записан на 9-дорожечную магнитную ленту [UNIT=TAPE].
2. Бобина идентифицирована именем ROSE [VOL=SER==ROSE].
3. Файл должен быть создан в данном задании [DISP==NEW].
4. Файл не помечен и является вторым файлом на этой ленте [LABEL=(2,NL)].
5. Записи имеют длину не более 80 литер. Записи выводятся бесформатными инструкциями WRITE и упаковываются как минимум по 40 записей в блоке [DCB=(RECFM==VBS,LRECL=84,BLKSIZE=3364)].

Необходимая DD-команда записывается в виде

```
//FT13F001 DD UNIT=TAPE,VOL=SER=ROSE,DISP=NEW,LABEL=(2,NL),
//            DCB=(RECFM=VBS,LRECL=84,BLKSIZE=3364)
```

Заметим, что данные, считываемые или выводимые бесформатными инструкциями ввода/вывода, должны быть в записях переменной длины. Поэтому мы не можем дать операнду RECFM значение FB.

Теперь перечислим все возможные значения операндов, встретившихся в рассмотренных выше примерах, и объясним их смысл.

UNIT=*тип-устройства*. Ключевое слово UNIT означает, что мы описываем тип используемого устройства. В некоторых системах UNIT=TAPE означает, что используется стандартное лентопротяжное устройство. Это лентопротяжное устройство может работать с 9-дорожечной лентой; плотность записи 800 или 1600 *bpi*. В вашем вычислительном центре стандартное назначение для 9-дорожечной магнитной ленты может быть другим. Например, UNIT=2400 и UNIT=3400-4 относятся к определенным типам лентопротяжек фирмы IBM.

VOL=SER=*имя-тома*. Ключевые слова VOL=SER определяют имя (серийный номер) тома ленты, т. е. бобины, которую оператор должен установить на свободную лентопротяжку. Имя ленты имеет длину до 8 литер и обычно присваивается персоналом вычислительного центра каждой бобине. Когда задание готово к выполнению, оператор получает от машины сообщение, что ленточный том с именем *имя-тома* должен быть установлен

на какое-либо лентопротяжное устройство. Пока лента не установлена, задание не выполняется.

DISP=*статус*. Ключевое слово DISP указывает статус (диспозицию) подлежащего обработке файла. У этого операнда могут быть три значения: NEW, OLD и MOD. NEW используют при создании нового файла. OLD, когда мы читаем или переписываем на другой том существующий файл. MOD задают в том случае, если желательно добавить записи к концу существующего файла. Заметим, что если файл не существует, то значение MOD эквивалентно NEW.

LABEL=*номер файла* или LABEL=(*номер файла*,NL). Ключевое слово LABEL используют для указания ленточного файла, подлежащего обработке. Первую форму применяют, если файл на ленте помечен; вторую — если файл не помечен (NL).

DSN=*имя файла*. Этот операнд необходимо указать, если ленточный файл помечен. Он задает имя файла длиной от 1 до 8 литер. Если файл создается в данном задании (DISP=NEW), то *имя-файла* будет записано в метку файла. Если файл уже создан (DISP=OLD), то *имя-файла* сравнивается с именем, которое ранее было записано в метке. Если имена не совпадают, то программа выполняться не будет. Поэтому операнд DSN обеспечивает некоторую защиту от постороннего вмешательства или непреднамеренной порчи ленточного файла.

DCB=(RECEM=*формат*, LRECL=*длина-записи*, BLKSIZE=*длина-блока*).

Этот операнд описывает, каким образом выводятся на ленту логические записи. Если RECFM=F, то все записи имеют фиксированную длину, равную *длине-записи*, и не сблокированы. В этом случае *длина-блока* должна быть такой же, как и длина логической записи. Если RECFM=FB, то все записи имеют фиксированный размер *длина-записи* и упакованы в блоки размера *длина-блока*, который должен быть кратным *длине-записи*. Если RECFM=V, то записи могут иметь переменную длину до (*длина-записи* — 4) и каждая запись выводится в виде отдельного блока. *Длина-блока* должна быть на 4 литеры больше, чем *длина-записи*. Указанное 4-литерное поле используют для управляющей информации. Если RECFM=VB, то длина записи может изменяться в пределах до (*длина-записи*—4). Такие логические записи упаковываются в блоки длиной, на 4 литеры меньшей, чем *длина-блока*. Значение *длины-записи* должно быть на 4 литеры больше, чем наибольшая возможная длина данных. Значение *длины-блока* должно быть на 4 литеры больше числа, кратного *длине-*

записи. Назначение RECFM=VBS в основном эквивалентно VB, но используется при бесформатном вводе/выводе.

Дополнительно к DD-командам в задании, включающем работу с магнитной лентой, могут потребоваться другие управляющие перфокарты — по одной для каждой бобины магнитной ленты. Форма написания и расположение этих команд в колоде целиком зависят от соглашений, принятых в вычислительном центре. Прежде чем работать с магнитной лентой, необходимо узнать, какие дополнительные JCL-карты для работы с лентой требуются в вашем вычислительном центре, какая информация на них должна быть задана и где такие карты располагаются в колоде.

4.3.2. Назначения устройствам в системе DEC-10

Фортран DEC-10 позволяет работать с непомеченными многофайловыми лентами. Каждый блок, выводимый на магнитную ленту, имеет фиксированную длину — обычно 128 слов (640 литер). Размеры этих блоков и их физические границы не связаны с размерами логических записей и их границами. Поэтому физический блок может содержать больше одной логической записи, а одна логическая запись может состоять из нескольких физических блоков.

Для указания границ логических записей применяются два метода. Один метод используется для форматных записей, другой — для бесформатных.

Форматную запись (выводимую форматной инструкцией WRITE) записывают с литерой конца строки, которую добавляет к записи сама система. Эта специальная литера сообщает о конце записи во время любых последующих операций чтения (см. рис. 4.3).

Бесформатную запись выводят с одним управляющим словом, предшествующим записи, и вторым, следующим за записью. Если запись содержит несколько блоков, то используется дополнительное управляющее слово для каждого блока, расположенного за первым. Управляющие слова содержат длину сегментов записи и применяются при считывании записей или при выполнении инструкции BACKSPACE (см. рис. 4.5).

При написании на Фортране инструкций ввода/вывода задают номера логических устройств каналов. К каждому такому каналу по умолчанию подключено некоторое устройство предопределенного класса. К фортранному каналу с номером 2 подключено физическое устройство ввода с перфокарт, к каналу 3 — АЦПУ, к каналу 5 — терминал, работающий в режиме разделения времени.

Обращений к каналам 2, 3 или 5 достаточно для ввода с перфокарт, вывода на печать или для работы с терминалом. Однако при обращении с помощью какого-либо номера канала к магнитной ленте необходимо выполнить определенные действия. Во-первых, попросить оператора установить данную ленту на какое-нибудь лентопротяжное устройство. Далее мы должны согласовать номер канала в программе на Фортране с этим физическим лентопротяжным устройством. Для этого используют команду MOUNT. Форма этой команды при обращении к ленте по номеру устройства канала *нк* имеет вид

.MOUNT MTA:*нк*/REELID:*имя-тома*/*доступ*

Команда MOUNT требует, чтобы бобина *имя-тома* была установлена на какое-нибудь лентопротяжное устройство. Оператор находит запрашиваемую ленту и устанавливает ее на какое-нибудь свободное устройство. Поскольку идентификационные номера бобин назначаются обслуживающим персоналом вычислительного центра, каждому такому номеру соответствует одна вполне определенная бобина магнитной ленты.

Значением *доступ* может быть RONLY или WENBL. Если задано RONLY (Read ONLY), то магнитная лента может быть только прочитана. Запись на ленту запрещена. Если задано WENBL (Write ENaBLe), то можно выполнять команды как чтения, так и записи на ленту. Этот операнд помогает предотвратить случайную порчу файлов на ленте.

Если необходимо установить несколько лент, то должна быть выполнена команда MOUNT для установки каждой ленты.

При установке лента ставится на начало. Начальная позиция на ленте может не соответствовать желаемой. Если, например, мы хотим обработать третий файл, то должны пропустить первых два файла. Для пропуска файлов выполняется команда SKIP. Ее формат

.SKIP *нк*: *n* FILES

Выполнение этой команды приводит к тому, что лента, подключенная к каналу *нк* предыдущей командой MOUNT, должна быть перемотана к началу $(n+1)$-го файла.

Команда SKIP может также иметь следующий формат:

.SKIP *нк*: EOT

Команда перематывает соответствующую магнитную ленту к концу последнего файла. Команду используют для добавления на ленту новых файлов.

После того как выполнение программы закончилось, для каждой установленной магнитной ленты следует задать команду

DISMOUNT. Форма этой команды:

.DISMOUNT *нк*:

Мы не дали здесь детального описания того, как именно включать команды MOUNT, SKIP и DISMOUNT, относящиеся к фортрановской программе. Это включение будет различным в зависимости от того, работаете ли вы в режиме *пакетной обработки* (представляете задание на перфокартах и получаете вывод на АЦПУ) или *разделения времени* (представляете на выполнение задание и получаете вывод на телетайп или дисплей). Подробная информация о том, как пользоваться описанными командами, приведена в руководствах DEC по режимам пакетной обработки и разделения времени. Следует проконсультироваться, каким из этих руководств пользоваться.

В любом случае для правильного использования магнитных лент нужно, чтобы команды MOUNT задавались перед выполнением программы, а команды DISMOUNT — после выполнения. Кроме того, во многих ВЦ требуется, чтобы перед любыми запросами MOUNT оператору была выдана команда PLEASE. Это делается для того, чтобы освободить лентопротяжное устройство. Прежде чем пытаться выполнить программу с обращением к магнитной ленте, следует узнать, какие специальные соглашения приняты в вашем ВЦ.

4.4. Устройства прямого доступа

Хотя магнитные ленты очень удобны для многих применений, существуют такие классы задач, для которых они не очень подходят. Рассмотрим, например, систему заказа авиационных билетов. Такая система должна иметь быстрый доступ к различным блокам данных в произвольном порядке. Поскольку магнитные ленты организованы так, что блок 2 может быть обработан только после блока 1, блок 3 — после блока 2 и т. д., ленты не годятся для обработки записей в произвольном порядке. Действительно, требуется как минимум несколько минут для того, чтобы прочесть блок, расположенный в конце ленты, если лента установлена на начало.

Запоминающие устройства, предназначенные для обеспечения быстрого доступа в произвольном порядке, называют *устройствами прямого доступа*. Наиболее широко используемые устройства этого типа — запоминающие устройства на магнитных дисках с *подвижными* и *фиксированными головками*.

Запоминающее устройство на магнитном диске представляет собой средство, которое с помощью головок чтения/записи обеспечивает доступ и запись данных на пакетах магнитных дисков *Пакет магнитных дисков* состоит из набора круглых пластин,

расположенных одна над другой (рис. 4.8). Каждая из этих пластин содержит большое число (обычно более 200) концентрических дорожек записи как на одной, так и на другой стороне пластины. Механизм чтения/записи содержит либо по одной головке чтения/записи на дорожку (диск с фиксированными головками), либо одну головку для каждой поверхности диска — одну головку чтения/записи, которая может быть выдвинута на любую выбранную дорожку (подвижные головки).

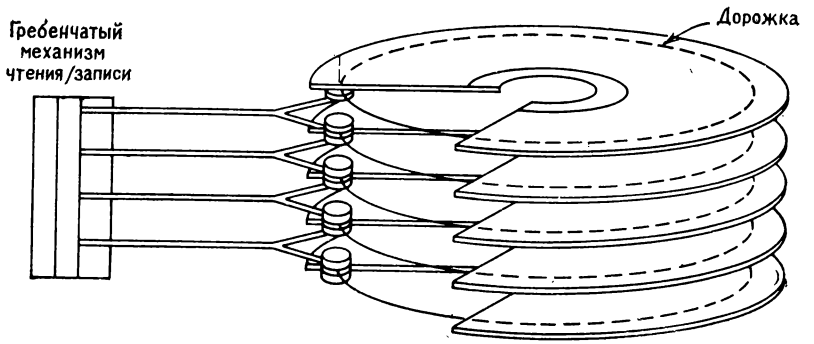


Рис. 4.8. Пакет дисков с подвижным механизмом чтения/записи.

Пакет дисков, устанавливаемый на устройство чтения/записи, вращается с некоторой фиксированной скоростью (обычно не менее 40 оборотов в секунду), так что каждый блок данных, записанный на дорожке диска, проходит под головкой чтения/записи один раз за оборот. Постоянная высокая скорость вращения плюс способность перемещать головки чтения/записи на любую дорожку обеспечивают быстрый доступ к блокам данных; обычно к любому заданному блоку можно получить доступ не более чем за пятую часть секунды, а среднее время доступа значительно меньше.

Прямой доступ к данным на пакете дисков выполняется легко потому, что каждый блок имеет свой собственный адрес. Этот адрес определяется расположением блока на дорожке, дорожки на поверхности пластины и пластины относительно других пластин.

Помимо прямого доступа, диски могут обеспечивать последовательный доступ. На самом деле в этом смысле они лучше лент, потому что можно обновлять любой блок на месте и записывать блоки любой длины, даже одиночную литеру. Все это достигается благодаря постоянному вращению дисков. Недостатки дисков по отношению к лентам обусловлены экономическими показателями. Дисковые устройства, обладающие более сложными техническими характеристиками, относительно дороже.

4.4.1. Характеристики некоторых дисковых устройств

Дисковые устройства с подвижными головками имеют большее применение, чем устройства с фиксированными головками. Это отчасти связано с тем, что они дешевле, а также вследствие того, что дисковые пакеты для устройств с фиксированными головками несъемные. В противоположность этому пакеты дисков с подвижными головками можно устанавливать и снимать, по существу, так же, как и магнитные ленты.

К обычно используемым дисковым устройствам с подвижными головками относятся устройства IBM 3330 и DEC RP03. Два общепринятых дисковых устройства с фиксированными головками — IBM 2305 и DEC RD10. Ниже приведены характеристики указанных четырех дисковых устройств.

Для машин	IBM	IBM	DEC	DEC
Производитель	IBM	IBM	Memorex	Burroughs
Тип	Подвижн.	Фиксир.	Подвижн.	Фиксир.
Номер модели	3330-1	2305-1	RP03	RD10
Максимальное число дисков в пакете	8	1	8	4
Емкость памяти (в млн. литер)	100	5.75	52	2.5
Время доступа (в миллисек.)				
Среднее	37.5	2.5	62.5	16.5
Максимальное	72	5	105	33
Скорость передачи (1000 литер/сек)	806	3000	333	385

Как видно из этой таблицы, следует отметить, что время доступа для дисковых устройств с фиксированными головками значительно меньше, чем для устройств с подвижными головками. Это связано с тем, что устройства с фиксированными головками имеют только *задержку вращения* (время, требуемое для подхода выбранного блока к головке чтения/записи). В противоположность этому устройства с подвижными головками имеют как задержку вращения, так и *задержку поиска* (время, требуемое для подвода головки чтения/записи к нужной дорожке).

4.4.2. Управление дисковой памятью в OS IBM 360-370

Дисковый пакет на устройстве чтения/записи IBM представляет для пользователя том и начинается с ряда дорожек, на которых записано оглавление тома (VTOC). Таблица-оглавление содержит по одному элементу (одной записи) на каждый записанный на пакете файл и по одному элементу на группу смежных незанятых дорожек. Запись для каждого файла указывает его

имя, формат его блоков и дорожки, отведенные под файл. Запись для каждой свободной области указывает адрес первой свободной дорожки и сообщает, сколько свободных дорожек входит в состав этой области.

VTOC имеет несколько назначений. Во-первых, пользователь, желающий записать на пакет новый файл, использует сведения о свободных областях. Если пакет содержит достаточную свободную область, то дорожки будут исключены из списка из свободных областей и будут отведены под новый файл, о чем в оглавлении будет сделана соответствующая запись. При уничтожении файла его дорожки вновь будут переданы в список свободного пространства.

Записи о файлах в оглавлении тома VTOC необходимы также в тех случаях, когда хотят обработать имеющийся набор данных. Они указывают, существует или нет набор данных на пакете, и если запись о наборе данных найдена, то она указывает адреса и форматы блоков набора данных.

4.4.3. Управление дисковой памятью в системе DEC-10

Блоки на дисковом пакете системы DEC-10 формируются во многом подобно блокам на магнитных лентах DEC-10. Данные записываются в блоках фиксированной длины, обычно по 128 слов. Фиксированный размер блока дает возможность распределять память для файлов в размерах, кратных размеру блока. Такое распределение не зависит от размеров дорожек диска.

Чтобы хранить сведения об используемой части пакета диска и свободной области, каждый пакет содержит таблицу распределения памяти, основное оглавление и набор оглавлений пользователей. Таблица распределения памяти содержит элементы, указывающие состояние каждого блока пакета (занят, свободен). Эта таблица нужна всякий раз, когда требуется область памяти для создания нового файла, или при освобождении области памяти вследствие уничтожения файла.

Основное оглавление включает по одному элементу на оглавление пользователей. Это сделано для того, чтобы находить нужное оглавление при поиске файлов, принадлежащих тому или иному пользователю. Оглавления пользователей содержат имя каждого файла пользователя и блоки, которые отведены этому файлу.

4.4.4. Последовательная обработка дисковых файлов

Данные могут быть записаны на диск или считаны с дискового файла тем же самым последовательным способом, как и с магнитной ленты. Действительно, форматы записей, используе-

мые для дисков, точно такие же, как и для ленты, а специальные команды BACKSPACE, ENDFILE и REWIND имеют логически то же самое действие, что и в магнитных лентах.

Указанная совместимость между последовательными файлами магнитных лент и дисков позволяет писать программы, которые работают с данными, записанными на любом магнитном носителе. В программах на Фортране не нужно делать никаких изменений. Однако для каждой системы при переходе от одного носителя к другому отдельные команды (например, команда MOUNT и DD-команды) должны быть определены по-своему.

4.4.5. Обработка дисковых файлов с произвольным доступом

Основное достоинство дисков по сравнению с лентами — их способность обеспечивать произвольный порядок обработки блоков с данными. Чтобы использовать эту возможность, в Фортране имеется режим ввода/вывода, называемый режимом *произвольного доступа*. С этим режимом связаны два новых типа инструкций. Кроме того, для работы в режиме произвольного доступа внесены незначительные изменения в инструкции READ и WRITE, применяемые для последовательной обработки.

Для осуществления ввода/вывода в режиме произвольного доступа вначале необходимо выполнить инструкцию, которая описывает подлежащий обработке файл. Эта инструкция, называемая DEFINE FILE, имеет следующую форму в Фортране IBM:

DEFINE FILE *нк* (*зап*, *разм*, L, *а-перем*)

и форму

CALL DEFINE FILE (*нк*, *разм*, *а-перем*, 'имя-файла.DAT')

в Фортране DEC-10. Параметры инструкций DEFINE FILE имеют следующее значение:

нк — фортрановский номер канала, назначенный для данного файла.

зап — Количество логических записей, которое должно быть ассоциировано с этим файлом. Каждая такая логическая запись имеет свой номер, который находится в диапазоне от 1 до *зап* (не определен в DEC-10).

разм — Длина каждой записи внутри файла. В Фортране IBM это количество литер. В Фортране DEC-10 это число слов для бесформатного В/В и число литер для форматного В/В.

а-перем — целая переменная, называемая *ассоциированной переменной*. Этой переменной присваивается значение 1

перед выдачей на устройство с *нк* любой команды ввода/вывода. После выполнения ввода/вывода значение *а-перем* всегда на 1 больше номера последней записи, которая была обработана.

Третий оставшийся параметр кодируется существенно различно для двух рассматриваемых нами вычислительных систем. В Фортране IBM этот параметр обозначен буквой L. (Существуют и другие значения этого параметра, но мы их не обсуждаем.) В Фортране DEC-10 на месте этого параметра нужно задать имя (в апострофах) подлежащего обработке файла произвольного доступа. Имя записывают в виде цепочки из 6 литер, за которой следует цепочка .DAT.

При выполнении инструкции DEFINE FILE на диске выделяется область памяти, которая подразделяется на записи *зап*, каждая длиной *разм*. Поскольку все записи фиксированной длины, система может легко вычислить, где расположена любая запись.

Инструкции READ и WRITE для файлов с произвольным доступом имеют следующую форму:

1. READ *(нк'n, формат) список*
2. READ *(нк'n) список*
3. WRITE *(нк'n, формат) список*
4. WRITE *(нк'n) список*

Инструкции 1 и 3 применяют для форматных записей, инструкции 2 и 4 — для бесформатных. Дополнительное поле *n* является целым выражением, значение которого представляет номер подлежащей обработке записи. После выполнения одной из приведенных инструкций значение ассоциированной переменной *(а-перем)* устанавливается в $n+1$. Поэтому инструкция

```
WRITE (2'15) (S(I), I=1, 20)
```

выводит запись из 20 слов, состоящую из двоичных представлений S(1), S(2), . . ., S(20), на дисковую область, ассоциированную с логической записью 15 на устройстве 2. После того как вывод завершен, ассоциированная переменная файла принимает значение 16.

Автоматическое приращение ассоциированной переменной файла с произвольным доступом может быть удобным, когда подлежащий обработке файл последовательного типа. Чтобы убедиться в этом, рассмотрим задачу чтения колоды перфокарт и записи ее в файл с произвольным доступом, ассоциированный с устройством 1. Полагая, что DSKPOS это ассоциированная переменная, достаточно написать следующие инструкции:

```
      DSKPOS = 1
10    READ (RDR,1000,END=99) (CARD(I), I=1,80)
1000  FORMAT (80A1)
      WRITE (1'DSKPOS,1000) (CARD(I), I=1,80)
      GO TO 10
```

До выполнения указанного цикла необходимо выполнить инструкцию DEFINE FILE. В Фортране IBM эта инструкция может быть задана как

DEFINE FILE 01 (200, 80, L, DSKPOS)

В Фортране DEC-10 можно было бы записать

CALL DEFINE FILE (1, 80, DSKPOS, 'FOR01.DAT')

В примере для IBM мы полагаем, что на диск должно быть записано не более 200 записей. В примере с DEC-10 таких ограничений нет.

При выполнении сегмента программы, состоящего из цикла — прочитать запись, обработать ее, вновь вернуться на чтение — было бы удобно иметь систему, которая подготавливалась к следующему чтению во время обработки текущей записи. Это может повысить эффективность программы, позволяя совмещать чтение с процессом обработки. В Фортране можно добиться такого совмещения выдачей инструкции FIND после каждой инструкции чтения. Инструкция FIND имеет следующий вид:

FIND *(нк'п)*

Ее выполнение приводит к тому, что система находит запись *п*. Эта запись не доступна для программы, пока не будет выдана последующая инструкция чтения, в которой опять указана запись *п*. Таким образом, эффективность программы можно повысить, написав вышеупомянутый цикл в виде последовательности: чтение текущей записи, нахождение следующей подлежащей обработке записи, обработка текущей записи, возвращение на чтение.

4.5. Назначение файлам устройств прямого доступа — примеры

4.5.1. Назначение устройствам прямого доступа в системе OS/JCL IBM

Так же как устройства ввода с перфокарт, АЦПУ и магнитные ленты, устройства прямого доступа ассоциируют с номерами каналов посредством DD-команд. В DD-командах используют ключевые слова UNIT, DISP, DSN и DCB, которые были уже нами рассмотрены, плюс одно новое ключевое слово SPACE.

Параметр LABEL, применяемый для магнитных лент, в устройствах прямого доступа не имеет смысла, а параметр VOL=SER хотя и имеет смысл, но мы его здесь не рассматриваем.

Параметр UNIT следует задавать так:

UNIT=SYSDA

Он указывает на работу с устройством прямого доступа (SYStem Direct Access device).

Параметры DSN и DCB задают так же, как и для ленты, с той разницей, что поле RECFM в DCB для файлов с произвольным доступом должно быть записано как RECFM=F.

Параметр DISP более сложный, чем для магнитных лент. Его записывают так:

DISP=(*статус*, *диспозиция*)

Здесь *статус* — NEW, OLD или MOD, а в поле *диспозиция* может быть задано DELETE, CATLG или KEEP. Указанные диспозиции означают следующее:

DELETE — уничтожить файл на диске после выполнения текущего задания. Все области, относящиеся к файлу, передаются операционной системе для перераспределения.

CATLG — сохранить файл, записав его имя и адрес в системный каталог. Последующие задания могут иметь доступ к этому набору данных просто по его имени.

KEEP — сохранить файл. Если файл уже был помещен в каталог в предыдущем задании, то он остается каталогизированным.

Параметр SPACE должен быть задан в том случае, если файл создается, т. е. если статус в DISP определен как NEW. Этот параметр сообщает о потребности файла в памяти. Он кодируется как

SPACE=(*блок*,(*нач*,*втор*),RLSE)

Значение *блок* определяет размеры физических блоков (так же, как BLKSIZE в параметре DCB); *нач* указывает, какое количество этих блоков необходимо первоначально; *втор* задает размер вторичной области, которая будет использована в том случае, если израсходована первоначально запрашиваемая область. Такая вторичная область может быть затребована до 15 раз. Поэтому максимальный размер файла равен *нач*+15**втор* блоков, каждый размером *блок*. Код RLSE говорит только, что любая неиспользованная при выполнении данного задания область будет возвращена системе.

Следующие примеры иллюстрируют кодирование DD-команд для различных задач:

Пример 1. Пусть для создания на диске последовательного файла с именем FORT мы предполагаем использовать устройство 18. Логические записи файла имеют длину 40 литер и сблокированы по 80 записей в блоке. Файл каталогизируется для дальнейшего применения.

```
//FT18F001 DD UNIT=SYSDA,DISP=(NEW,CATLG),DSN=FORT,
//            DCB=(RECFM=FB,LRECL=40,BLKSIZE=3200),
//            SPACE=(3200,(50,10),RLSE)
```

Параметр SPACE отводит область памяти на диске под 50 блоков. Если все блоки использованы, размер области возрастает до 60, затем до 70 и т. д. максимально до 200 блоков.

Пример 2. Пусть для обработки файла, созданного в примере 1, мы предполагаем использовать устройство 8. Файл должен быть сохранен после завершения данного задания.

```
//FT08F001 DD DISP=(OLD,KEEP),DSN=FORT
```

Пример 3. Для обработки файла, созданного в примере 1, мы предполагаем использовать устройство 12. После завершения данного задания, файл должен быть уничтожен.

```
//FT12F001 DD DISP=(OLD,DELETE),DSN=FORT
```

4.5.2. Назначение устройствам прямого доступа в системе DEC-10

Любой пользователь может получить область памяти на дисках в режиме прямого доступа (до некоторого максимального размера области, задаваемого администратором). В отличие от системы IBM 360-370 нет необходимости предварительно запрашивать область для каждого файла. На самом деле при выполнении инструкции CALL DEFINE FILE происходит только назначение устройства и наименование файла.

Имя последовательного дискового файла, связанного с каналом *нк*, всегда представлено в виде FOR*нк*.DAT. Назначение дискового устройства происходит автоматически для номеров каналов 1, 20, 21, 22, 23 и 24. Назначение диска другим номерам каналов происходит посредством команды ASSIGN следующим образом:

.ASSIGN DSK: *нк*

После завершения выполнения программы все созданные при данном выполнении файлы остаются на диске. Файлы, которые больше не нужны, могут быть уничтожены командой

.DELETE *имя-файла*.DAT

(*Замечание*. Тем, кто интересуется подробностями организации ввода/вывода в системе DEC-10, следует ознакомиться с инструкцией OPEN Фортрана DEC-10. Она дает больше возможностей, чем команда ASSIGN.)

4.6. Пример. Система поиска книг по ключевым словам

Рассмотрим задачу поиска в библиотеке всех книг, посвященных некоторой научно-технической области. Один из наиболее часто используемых методов — сканирование каталогов библиотеки по всем заглавиям, содержащим некоторые ключевые слова. Например, при поиске книг, касающихся языков программирования, мы будем, вероятно, проверять заглавия книг, в которых содержатся слова Алгол, Кобол, Фортран, ПЛ/I и др. Ясно, что такие операции лучше выполняют вычислительные машины, чем люди, поэтому мы построим программу поиска по ключевым словам.

Предполагаемая программа имеет два входных файла. Первый, ленточный или дисковой последовательный файл содержит заглавия всех книг, принадлежащих библиотеке. Логические записи внутри этого файла представляют собой записи, каждая из которых имеет длину 70 литер и содержит заглавие одной книги.

Второй входной файл вводится с перфокарт. Колода перфокарт состоит из двух частей. Она начинается одной или серией из нескольких перфокарт, каждая из которых содержит ключевое слово в колонках с 1 по 8. Окончание ключевых слов сигнализируется перфокартой, содержащей звездочку (*), пробитую в колонке 1. Следом за перфокартой со звездочкой располагают набор перфокарт команд поиска. Каждая из этих перфокарт содержит одноразрядный код в колонке 1. Некоторые из этих перфокарт содержат ключевые слова, пробиваемые в колонках с 3 по 10. Разрешенные коды команд и их значения следующие:

1. Установить систему поиска в исходное состояние, когда ни одна книга еще не выбрана для вывода на печать.

2. Требует ключевого слова. По этой команде все заглавия, содержащие данное ключевое слово, будут добавлены к списку уже отобранных заглавий. Так, при выполнении последовательности команд

```
1
2 ПРОГРАММА
2 ПРОГРАММЫ
```

будут выбраны все заглавия, которые содержат слово ПРОГРАММА или ПРОГРАММЫ.

3. Требует ключевого слова. Эта команда позволяет включить в список заглавий только те ранее отобранные заглавия, которые содержат данное новое ключевое слово. Поэтому при выполнении последовательности команд

   1
   2 ПРОГРАММА
   3 СТИЛЬ

   будут выбраны все заглавия, которые содержат оба слова ПРОГРАММА и СТИЛЬ.

4. Отрицание списка ранее выбранных заглавий так, что теперь будут выбраны только те заглавия, которые ранее не были выбраны. Поэтому команды

   1
   2 ПРОГРАММА
   4

   отберут все заглавия, которые не содержат слово ПРОГРАММА.

5. Напечатать отобранные заглавия. Эта команда печатает заглавия всех книг, которые были выбраны в текущем задании. Поэтому при выполнении команд

   1
   2 РАЗДЕЛЕНИЕ
   2 ОТРЕЗКИ
   3 ВРЕМЕНИ
   4
   5

   будут напечатаны заглавия, которые не содержат пары слов (РАЗДЕЛЕНИЕ ВРЕМЕНИ) или (ОТРЕЗКИ ВРЕМЕНИ).

Две основные структуры данных, используемые для выполнения приведенных пяти команд поиска, представляют собой двумерный логический массив TTLKEY и логический список SELECT. Во время выполнения фазы программы, в которой первоначально вводятся заглавия, в каждом заглавии просматривают-

ся содержащиеся в нем ключевые слова. Если K-е ключевое слово содержится в I-м заглавии, то элементу TTLKEY(I,K) будет присвоено значение .TRUE.; в противном случае .FALSE. .

После того как началась программная фаза, в которой обрабатываются команды поиска, список SELECT определяет, какие заглавия в текущем задании выбираются для печати. Значение SELECT(I) устанавливается в .TRUE. в том случае, если выбрано I-е заглавие. Таким образом, задача подпрограммы поиска состоит единственно в том, чтобы обновить каждый элемент SELECT на основании определенной команды и значений, найденных в TTLKEY. Поэтому, например, команда

3 ФОРТРАН

где ФОРТРАН является K-м ключевым словом, реализуется посредством инструкции

```
SELECT(I)=SELECT(I) .AND. TTLKEY(I, K)
```

для I в диапазоне от 1 до количества хранящихся заглавий. Команда печати (5) выводит на печать I-е заглавие только в случае, если SELECT(I) имеет значение .TRUE. .

Программа, реализующая данную информационно-поисковую систему, приведена на рис. 4.9.

```
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* СИСТЕМА ПОИСКА КНИГ ПО КЛЮЧЕВЫМ СЛОВАМ.                              *
C*  ДАННАЯ ПРОГРАММА ВВОДИТ НАБОР ЗАГЛАВИЙ КНИГ И КЛЮЧЕВЫХ*
C*  СЛОВ, ИСПОЛЬЗУЕМЫХ ДЛЯ ИНДЕКСАЦИИ КНИГ. ЗАТЕМ ВВОДИТСЯ*
C*  ИНФОРМАЦИЯ, УСТАНАВЛИВАЮЩАЯ ПО КЛЮЧЕВЫМ СЛОВАМ, К КАКИМ*
C*  КНИГАМ ПОЛЬЗОВАТЕЛЬ ХОЧЕТ ИМЕТЬ ДОСТУП. СПИСКИ СООТВЕТ-*
C*  СТВУЮЩИХ ЗАГЛАВИЙ КНИГ ВЫВОДЯТСЯ ПО ТРЕБОВАНИЮ НА ПЕЧАТЬ.*
```

Рис. 4.9. Система поиска книг по ключевым словам. (В главной программе: **50**: «Автоматизированная система поиска книг по ключевым словам»; **60**: «Слова, которыми можно пользоваться при поиске».; **70**: «Отладка. Матрица, созданная при выполнении данного задания»; **303**: «Ошибка: недозволенный код команды»; **331**: «Ошибка: недозволенное ключевое слово»; **361**: «Заглавия, найденные по последнему набору команд»; **401**: «Все запросы обработаны»; **7001**: «Ошибка: нет ключевых слов»; **7501**: «Ошибка: нет заглавий»; **8001**: «Ошибка: неожиданный конец данных». В подпрограмме GETKEY: **50**: «Трассировка — вход в GETKEY»; **70**: «Трассировка — номер ключевого слова»; **130**: «Трассировка — выход из GETKEY»; **8001**: «Трассировка — выход из GETKEY — конец данных». В подпрограмме GETTTL: **50**: «Трассировка — вход в GETTTL; **70**: «Отладка — номер заглавия»; **210**; «Трассировка — выход из GETTTL». В подпрограмме XTRACT: **50**: «Трассировка — вход в XTRACT»; **301**: «Трассировка — выход из XTRACT». В подпрограмме NEXTWD: **50**: «Трассировка — вход в NEXTWD»; **121**: «Трассировка — выход из NEXTWD, новых слов не найдено»; **350**: «Трассировка — выход из NEXTWD». В подпрограмме KEYLUK: **50** и **201** — трассировка входа и выхода).

```
C* ФОРМАТ ВВОДА (ФАЙЛ ОБРАЗОВ ПЕРФОКАРТ):                          *
C*  НАБОР ПЕРФОКАРТ, КАЖДАЯ ИЗ КОТОРЫХ СОДЕРЖИТ КЛЮЧЕВОЕ*
C*     СЛОВО В КОЛОНКАХ 1-8.                                        *
C*  КАРТА МАРКЕР КОНЦА ДАННЫХ, СОДЕРЖАЩАЯ * В КОЛОНКЕ 1.            *
C*  НАБОР ЗАПРОСОВ НА ПОИСК, КАЖДЫЙ В ФОРМЕ:                        *
C*     КОЛОНКА 1 — КОД КОМАНДЫ (1, 2, 3, 4 ИЛИ 5);                 *
C*     КОЛОНКИ 3—10 — КЛЮЧЕВЫЕ СЛОВА (ИСПОЛЬЗУЮТСЯ ТОЛЬКО*
C*                    В ТОМ СЛУЧАЕ, ЕСЛИ КОД КОМАНДЫ 2 ИЛИ 3)*
C* ФОРМАТ ВВОДА (ЛЕНТОЧНЫЙ ИЛИ ДИСКОВЫЙ ПОСЛЕДОВАТЕЛЬНЫЙ*
C*                                                              ФАЙЛ):*

C*  НАБОР ЗАГЛАВИЙ КНИГ (ЗДЕСЬ ДЛИНА ЗАГЛАВИЯ — 70 ЛИТЕР)          *
C*                                                                  *
C* ИНТЕРПРЕТИРУЕМЫЕ КОМАНДЫ ПОИСКА:                                 *
C*  1 — ВОССТАНОВИТЬ ИСХОДНЫЕ ЗНАЧЕНИЯ В СПИСКЕ SELECT.            *
C*  2 — ДОБАВИТЬ К СПИСКУ SELECT ВСЕ ЗАГЛАВИЯ С ДАННЫМ КЛЮЧЕ-*
C*     ВЫМ СЛОВОМ.                                                  *
C*  3 — ВЫЧЕРКНУТЬ ИЗ СПИСКА SELECT ЗАГЛАВИЯ, НЕ СОДЕРЖАЩИЕ*
C*     ДАННОЕ КЛЮЧЕВОЕ СЛОВО.                                       *
C*  4 — ОТРИЦАНИЕ РАНЕЕ ВЫБРАННОГО СПИСКА, ЧТОБЫ ВЫБРАТЬ ВСЕ*
C*     ЗАГЛАВИЯ, ОТСУТСТВОВАВШИЕ В ПРЕДЫДУЩЕМ СПИСКЕ.              *
C*  5 — НАПЕЧАТАТЬ ЗАГЛАВИЯ ВСЕХ ВЫБРАННЫХ КНИГ.                    *
C*                                                                  *
C* РЕАЛИЗУЕМЫЙ АЛГОРИТМ:
C*  A) КЛЮЧЕВЫЕ СЛОВА ВВОДЯТСЯ И ЗАПОМИНАЮТСЯ В СПИСКЕ              *
C*     KEYTAB.                                                      *
C*  B) ЗАГЛАВИЯ ВВОДЯТСЯ И ЗАПОМИНАЮТСЯ В ФАЙЛЕ С ПРОИЗВОЛЬ-*
C*     НЫМ ДОСТУПОМ.                                                *
C*  C) ПРИ СЧИТЫВАНИИ В КАЖДОМ ЗАГЛАВИИ ПРОСМАТРИВАЮТСЯ СО-*
C*     ДЕРЖАЩИЕСЯ В НЕМ КЛЮЧЕВЫЕ СЛОВА. ЕСЛИ ЭТО I-Е ЗАГЛА-*
C*     ВИЕ И ОНО СОДЕРЖИТ K-Е КЛЮЧЕВОЕ СЛОВО, ТО ЭЛЕМЕНТУ*
C*     TTLKEY (I, K) ПРИСВАИВАЕТСЯ ЗНАЧЕНИЕ. TRUE.                 *
C*  D) УСТАНАВЛИВАЕТСЯ В ИСХОДНОЕ СОСТОЯНИЕ СПИСОК SELECT*
C*     ДЛЯ УКАЗАНИЯ ТОГО, ЧТО НИ ОДНО ЗАГЛАВИЕ ЕЩЕ НЕ ВЫБРА-*
C*     НО: (SELECT(I)=.FALSE., ДЛЯ ВСЕХ I).                         *
C*  E) ВВОДЯТСЯ КОМАНДЫ ПОИСКА. ВЫПОЛНЯЮТСЯ ДЕЙСТВИЯ:              *
C*     1 — SELECT(I)=.FALSE., ДЛЯ ВСЕХ I;                           *
C*     2 — SELECT(I)=SELECT(I). OR. TTLKEY (I,K)                    *
C*        ДЛЯ ВСЕХ I, ГДЕ K ЭТО ИНДЕКС КЛЮЧЕВОГО СЛОВА, УКА-*
C*        ЗАННОГО В ДАННОЙ КОМАНДЕ;                                 *
C*     3 — SELECT(I)=SELECT(I) .AND. TTLKEY (I,K)                   *
C*        ДЛЯ ВСЕХ I, K (КАК ДЛЯ КОМАНДЫ 2);                        *
C*     4 — SELECT(I) =.NOT.SELECT(I) ДЛЯ ВСЕХ I;                    *
C*     5 — ЕСЛИ SELECT(I)=.TRUE., НАПЕЧАТАТЬ                        *
C*        I-Е ЗАГЛАВИЕ.                                             *
C*                                                                  *
C* ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                             *
C*  KEYTAB  — ТАБЛИЦА КЛЮЧЕВЫХ СЛОВ; CHARACTER*8;                  *
C*            РАЗМЕР 50;                                            *
C*  KEYMAX  — МАКСИМАЛЬНЫЙ РАЗМЕР KEYTAB; ЗДЕСЬ                    *
C*            РАЗМЕР 50;                                            *
C*  KEYNUM  — ДЕЙСТВИТЕЛЬНОЕ КОЛИЧЕСТВО КЛЮЧЕВЫХ СЛОВ;             *
C*  TTLKEY  — МАТРИЦА ПОИСКА, LOGICAL, РАЗМЕР 100*50;              *
C*  TTLMAX  — МАКСИМАЛЬНОЕ КОЛИЧЕСТВО ЗАГЛАВИЙ; ЗДЕСЬ 100;         *
C*  TTLNUM  — ДЕЙСТВИТЕЛЬНОЕ КОЛИЧЕСТВО ЗАГЛАВИЙ;                  *
C*  TTLWTH  — ДЛИНА В ЛИТЕРАХ КАЖДОГО ЗАГЛАВИЯ;                    *
C*            ЗДЕСЬ 70;                                             *
C*  SELECT  — ТЕКУЩИЕ ВЫБРАННЫЕ ЗАГЛАВИЯ, LOGICAL,                 *
C*            РАЗМЕР 50;                                            *
C*  TRACE   — ЕСЛИ TRUE, ТРАССИРОВКА ВСЕХ ОБРАЩЕНИЙ К ПОД-*
C*            ПРОГРАММАМ И ВЫХОДОВ, LOGICAL;                        *
C*  DEBUG   — ЕСЛИ TRUE, ПЕЧАТЬ ВСЕХ ВХОДНЫХ ДАННЫХ И ДАМП*
C*            TTLKEY,(LOGICAL).                                     *
C*                                                                  *
C* ИСПОЛЬЗУЕМЫЕ ПОДПРОГРАММЫ:                                       *
C*  XTRACT  — ПРИСВОИТЬ ЗНАЧЕНИЯ TTLKEY В СООТВЕТСТВИИ С*
C*  НАЙДЕННЫМИ СЛОВАМИ В ТЕКУЩЕМ ЗАГЛАВИИ;                          *
C*  KEYLUK  — ПРОВЕРИТЬ, ЯВЛЯЕТСЯ ЛИ ЗАДАННОЕ СЛОВО КЛЮ-*
C*            ЧЕВЫМ;                                                *
```

Продолжение Рис. 4.9.

```
C*  GETKEY  — ВВЕСТИ И ЗАПОМНИТЬ НАБОР КЛЮЧЕВЫХ СЛОВ;          *
C*  GETTTL  — ВВЕСТИ И ЗАПОМНИТЬ ЗАГЛАВИЯ;                     *
C*  NEXTWD — ВОЗВРАТИТЬ СЛЕДУЮЩЕЕ СЛОВО В ЗАГЛАВИИ            *
C*            ПОСЛЕ КОЛОНКИ 'COL'.                             *
C*                                                             *
C*  ПРОГРАММИСТ; ———            ДАТА: ———                      *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER DSKPTR, KEYNDX, COMAND, I, J
      LOGICAL SELECT(50)
      CHARACTER KEYWD*8
C       ЗАМЕЧАНИЕ: ОБЩИЕ ПЕРЕМЕННЫЕ ИНИЦИАЛИЗИРУЮТСЯ В BLOCK DATA
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
C
C       ВЫДЕЛИТЬ ОБЛАСТЬ И НАЗНАЧИТЬ УСТРОЙСТВО 'DIRECT' ФАЙЛУ
C       С ПРОИЗВ. ДОСТУПОМ. ПОКАЗАНЫ ФОРМЫ ДЛЯ IBM 360-370 И DEC-10
      DEFINE FILE 12 (100, 70, L, DSKPTR)
C-----CALL DEFINE FILE (12, 70, DSKPTR, 'TITLES.DAT')
C
C       ПЕРЕД НАЧАЛОМ ОБРАБОТКИ ВЫВЕСТИ ЗАГОЛОВОК РАСПЕЧАТКИ
      WRITE (PRT,50)
50    FORMAT ('1', T20, 'AUTOMATED KEYWORD BOOK RETRIEVAL SYSTEM' /
     1        '+', T20, '_________ _______ ____ _________ ______' /// )
C
C       ВВЕСТИ И ЗАПОМНИТЬ КАЖДОЕ КЛЮЧЕВОЕ СЛОВО
      CALL GETKEY (&8000)
C
C       ЕСЛИ КЛЮЧ НЕ НАЙДЕН, ПРЕКРАТИТЬ РАБОТУ
      IF (KEYNUM .EQ. 0) GO TO 7000
C
C       НАПЕЧАТАТЬ СПИСОК ПРАВИЛЬНЫХ КЛЮЧЕВЫХ СЛОВ
      WRITE (PRT,60) (KEYTAB(I), I = 1, KEYNUM).
60    FORMAT (T17, 'KEYWORDS WHICH MAY BE USED FOR RETRIEVAL ARE:'
     1        /    (1X, 8A10) )
C
C       ВВЕСТИ ЗАГЛАВИЯ, ЗАПИСАТЬ НА ДИСК, СОЗДАТЬ МАТРИЦУ ПОИСКА
      CALL GETTTL
C
C       ЕСЛИ ЗАГЛАВИЙ НЕТ, ПРЕКРАТИТЬ РАБОТУ
      IF (TTLNUM .EQ. 0) GO TO 7500
C
C       В ЛЮБОМ СЛУЧАЕ НАЧИНАТЬ ПОИСК, КАК ЕСЛИ БЫ БЫЛА
C       НАЙДЕНА КОМАНДА СБРОСА (КОД=1).
C       ОНА ИНИЦИАЛИЗИРУЕТ SELECT
      IF (.NOT. DEBUG) GO TO 320
      WRITE (DBG,70)
70    FORMAT (' *** DEBUG *** RETRIEVAL MATRIX CREATED FOR THIS RUN')
      DO 100 I = 1, TTLNUM
         WRITE (DBG,80) I, (TTLKEY(I,J), J = 1, KEYNUM)
```

Продолжение Рис. 4.9.

```
80        FORMAT (' TITLE', I3, ':', (T11, 50L1) )
100     CONTINUE
        GO TO 320
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C ФАЗА ПОИСКА: ВСЕ КОМАНДЫ ПОИСКА ОБРАБАТЫВАЮТСЯ ЗДЕСЬ                    *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
300     READ (RDR,301,END=400) COMAND, KEYWD
301     FORMAT (I1, 1X, A8)
        IF (DEBUG) WRITE (DBG,302) COMAND, KEYWD
302     FORMAT (' *** DEBUG *** RETRIEVAL CODE', I2, ' KEYWORD ', A8)
C
C         УБЕДИМСЯ, ЧТО КОМАНДА ВЕРНАЯ
        IF ((COMAND .GE. 1) .AND. (COMAND .LE. 5)) GO TO 310
        WRITE (PRT,303) COMAND
303     FORMAT (' *** ERROR *** - ILLEGAL COMMAND CODE=', I1)
        GO TO 300
C
C         ПЕРЕХОД НА ОБРАБОТКУ КОНКРЕТНОЙ КОМАНДЫ
310     GO TO (320,330,330,350,360), COMAND
C
C         КОД 1 - ВОССТАНОВИТЬ СПИСОК SELECT
320     DO 325 I = 1, TTLNUM
           SELECT(I) = .FALSE.
325     CONTINUE
        GO TO 300
C
C         КОДЫ 2 И 3 ТРЕБУЮТ ПРАВИЛЬНОГО КЛЮЧЕВОГО СЛОВА
330     KEYNDX = KEYLUK(KEYWD)
        IF (KEYNDX .GT. 0) GO TO 332
C


C         КЛЮЧ НЕВЕРНЫЙ - СООБЩЕНИЕ ОБ ОШИБКЕ
        WRITE (PRT,331) KEYWD
331     FORMAT (' *** ERROR *** - ILLEGAL KEYWORD SPECIFIED:', A8)
        GO TO 300
332     IF (COMAND .EQ. 3) GO TO 340
C
C         КОД 2 - ДОБАВИТЬ ЭЛЕМЕНТЫ К СПИСКУ SELECT
        DO 335 I = 1, TTLNUM
           SELECT(I) = SELECT(I) .OR. TTLKEY(I,KEYNDX)
335     CONTINUE
        GO TO 300
C         КОД 3 - ЛОГИЧ. УМНОЖ. SELECT НА ЭЛЕМЕНТЫ,
C         СООТВЕТСТВУЮЩИЕ ЗАДАННЫМ КЛЮЧЕВЫМ СЛОВАМ
340     DO 345 I = 1, TTLNUM
           SELECT(I) = SELECT(I) .AND. TTLKEY(I,KEYNDX)
345     CONTINUE
        GO TO 300
C
C         КОД 4 - ЛОГИЧ. ОТРИЦАНИЕ СПИСКА SELECT
350        355 I = 1, TTLNUM
           SELECT(I) = .NOT. SELECT(I)
355     CONTINUE
        GO TO 300
C
C         КОД 5 - ВЫВЕСТИ ВЫБРАННЫЕ ЗАГЛАВИЯ
360     WRITE (PRT,361)
361     FORMAT ('0 TITLES RETRIEVED BY LATEST COMMAND SEQUENCE ARE:' /)
        DO 365 I = 1, TTLNUM
           IF (.NOT. SELECT(I)) GO TO 365
           READ (DIRECT'I) TITLE
           WRITE (PRT,362) I, TITLE
362        FORMAT (' TITLE', I4, ' :', A70)
365     CONTINUE
        GO TO 300
C
C*************************************************************
C**** ВСЕ ЗАПРОСЫ НА ПОИСК ВЫПОЛНЕНЫ - НОРМАЛЬНОЕ ОКОНЧАНИЕ ****
400     WRITE (PRT,401)
401     FORMAT ('0*** ALL PROCESSING COMPLETED ***')
        STOP
```

Продолжение Рис. 4.9.

```
C
C*************************************************************
C****  СООБЩЕНИЯ ОБ ОКОНЧАНИИ ПО ОШИБКЕ                  ****
C
C       НЕТ КЛЮЧЕВОГО СЛОВА
7000  WRITE (PRT,7001)
7001  FORMAT (' *** ERROR *** - NO KEYWORDS FOUND')
      STOP
C
C       НЕТ ЗАГЛАВИЯ
7500  WRITE (PRT,7501)
7501  FORMAT (' *** ERROR *** - NO TITLES FOUND')
      STOP
C
C       НЕОЖИДАННЫЙ КОНЕЦ ФАЙЛА
8000  WRITE (PRT,8001)
8001  FORMAT (' *** ERROR *** - UNEXPECTED END OF DATA')
      STOP
      END

      SUBROUTINE GETKEY (*)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВВЕСТИ И ЗАПОМНИТЬ ВСЕ КЛЮЧЕВЫЕ СЛОВА В ТАБЛИЦЕ KEYTAB, ЕСЛИ НЕТ   *
C* КАРТЫ (*) ОКОНЧАНИЯ КЛЮЧЕВЫХ СЛОВ, ТО RETURN1 И ОСТАНОВ            *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      INTEGER I
      CHARACTER KEYWD*8, KEYCH*1(8)
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      EQUIVALENCE (KEYWD,KEYCH(1))
      IF (TRACE) WRITE (TRC,50)
50    FORMAT (' *** TRACE *** ENTERING GETKEY')
      DO 100 I = 1, KEYMAX
         READ (RDR,60,END=8000) KEYWD
60       FORMAT (A8)
         IF (KEYCH(1) .EQ. '*') GO TO 120
         IF (DEBUG) WRITE (DBG,70) I, KEYWD
70       FORMAT (' *** DEBUG *** KEYWORD NUMBER', I3, ' IS ', A8)
         KEYTAB(I) = KEYWD
100   CONTINUE
C
C       СУЩЕСТВУЕТ КАК МИНИМУМ KEYMAX ЭЛЕМЕНТОВ
      I = KEYMAX+1
C
C       УБЕДИМСЯ, ЧТО НАШЛИ * , ИГНОРИРОВАТЬ ВСЕ СВЕРХ KEYMAX
110   READ (RDR,60,END=8000) KEYWD
      IF (KEYCH(1) .NE. '*') GO TO 110
120   KEYNUM =I-1
      IF (TRACE) WRITE (TRC,130)
130   FORMAT (' *** TRACE *** EXITING GETKEY - NORMAL RETURN')
      RETURN
C
C       НЕОЖИДАЕМЫЙ КОНЕЦ ФАЙЛА, СООБЩЕНИЕ ОБ ОШИБКЕ
8000  IF (TRACE) WRITE (TRC,8001)
8001  FORMAT (' *** TRACE *** EXITING GETKEY - END OF DATA RETURN')
      RETURN1
      END

      SUBROUTINE GETTTL
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ВВЕСТИ ЗАГЛАВИЯ. ЗАПИСАТЬ ИХ В ФАЙЛ С ПРОИЗВОЛЬНЫМ ДОСТУПОМ.   *
C* В КАЖДОМ ЗАГЛАВИИ ОПРЕДЕЛИТЬ, КАКОЕ ОНО СОДЕРЖИТ КЛЮЧЕВОЕ       *
C* СЛОВО. ЕСЛИ I-ОЕ ЗАГЛАВИЕ СОДЕРЖИТ K-ОЕ КЛЮЧЕВОЕ СЛОВО,        *
C* ТО УСТАНОВИТЬ TTLKEY(I,K) В TRUE                                *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ: XTRACT                                *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Продолжение Рис. 4.9.

```
      INTEGER I
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      IF (TRACE) WRITE (TRC,50)
50    FORMAT (' *** TRACE *** ENTERING GETTTL')
      DO 100 I = 1, TTLMAX
         READ (MASTER,60,END=200) TITLE
60       FORMAT (A70)
         IF (DEBUG) WRITE (DBG,70) I, TITLE
70       FORMAT (' *** DEBUG *** TITLE NUMBER', I4, ' IS:', A70)
         WRITE (DIRECT'I) TITLE
         CALL XTRACT (I)
100   CONTINUE
C          НЕ БОЛЬШЕ TTLMAX ЗАГЛАВИЙ, ИГНОРИРОВАТЬ
C          ЛЮБОЕ СВЕРХ ЭТОГО КОЛИЧЕСТВА
      I = TTLMAX+1
200   TTLNUM = I-1
      IF (TRACE) WRITE (TRC,210)
210   FORMAT (' *** TRACE *** EXITING GETTTL')
      RETURN
      END


      SUBROUTINE XTRACT (I)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИЗВЛЕЧЬ ВСЕ СЛОВА ИЗ I-ГО ЗАГЛАВИЯ. ДЛЯ КАЖДОГО ВСТРЕТИВШЕГОСЯ    *
C* КЛЮЧЕВОГО СЛОВА  УСТАНОВИТЬ ЗНАЧЕНИЕ СООТВЕТСТВУЮЩЕГО ЭЛЕМЕНТА   *
C* В МАТРИЦЕ ПОИСКА TTLKEY                                            *
C* ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ: NEXTWD, KEYLUK                            *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER COL, I, J
      CHARACTER KEYWD*8
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      IF (TRACE) WRITE (TRC,50) I
50    FORMAT (' *** TRACE *** ENTERING XTRACT, I=', I3)
C
C       ПОКА НЕТ КЛЮЧЕВЫХ СЛОВ
      DO 100 J = 1, KEYNUM
         TTLKEY(I,J) = .FALSE.
100   CONTINUE
C
C       НАЧИНАЯ С КОЛОНКИ 1, СКАНИРУЕМ ПЕРФОКАРТУ
      COL = 1
C
C       НАХОДИМ СЛЕДУЮЩЕЕ СЛОВО, ЕСЛИ НЕТ, ПЕРЕХОД НА 300
200   CALL NEXTWD (&300, COL, KEYWD)
      KEYNDX = KEYLUK(KEYWD)
      IF (KEYNDX .EQ. 0) GO TO 200
C
C       НАЙДЕНО КЛЮЧЕВОЕ СЛОВО — ОТМЕТИМ В МАТРИЦЕ
      TTLKEY(I,KEYNDX) = .TRUE.
      GO TO 200
C
C       ЗАГЛАВИЕ ПОЛНОСТЬЮ ПРОСМОТРЕНО — ВОЗВРАТ
300   IF (TRACE) WRITE (TRC,301)
301   FORMAT (' *** TRACE *** EXITING XTRACT')
      RETURN
      END
```

Продолжение Рис. 4.9.

```
      SUBROUTINE NEXTWD (*, COL, WORD)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* НАЧИНАЯ С КОЛОНКИ 'COL', НАХОДИМ СЛЕДУЮЩЕЕ СЛОВО В ДАННОМ            *
C* ЗАГЛАВИИ. ЕСЛИ ОНО БОЛЬШЕ 8 ЛИТЕР, УСЕКАЕМ ЕГО ДО 8.                *
C* 'COL' УКАЗЫВАЕТ НА ЛИТЕРУ ПОСЛЕ НОВОГО СЛОВА                         *
C* НОВОЕ СЛОВО ЗАПИСАНО В ПАРАМЕТР 'WORD'                               *
C* ЕСЛИ СЛЕДУЮЩЕГО СЛОВА НЕТ — RETURN 1                                 *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      INTEGER COL, I, J
      CHARACTER WORD*8, CHARS*1(8), WD*8, TTLCH*1(70)
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      EQUIVALENCE (WD,CHARS(1)), (TITLE,TTLCH(1))
      IF (TRACE) WRITE (TRC,50) COL
50    FORMAT (' *** TRACE *** ENTERING NEXTWD, COL=', I2)
C
C        ЕСЛИ УЖЕ ПРОШЛИ КОНЕЦ ЗАГЛАВИЯ, ВОЗВРАТ
      IF (COL .GT. TTLWTH) GO TO 120
C
C        ИЩЕМ СЛЕДУЮЩУЮ ЛИТЕРУ, НЕ ПРОБЕЛ
      DO 100 I = COL, TTLWTH
         IF (TTLCH(I) .NE. ' ') GO TO 150
100   CONTINUE
C
C        ОСТАТОК КАРТЫ — ПРОБЕЛЫ
120   IF (TRACE) WRITE (TRC,121)
121   FORMAT (' *** TRACE *** EXITING NEXTWD, NO NEW WORD FOUND')
      RETURN1


C        СЛОВО НАЙДЕНО,
C        ОЧИСТИТЬ ОБЛАСТЬ ПРОБЕЛАМИ
150   WD = ' '
      DO 200 J = 1, 8
         IF ((I+J-1) .GT. TTLWTH) GO TO 250
         IF (TTLCH(I+J-1) .EQ. ' ') GO TO 250
         CHARS(J) = TTLCH(I+J-1)
200   CONTINUE
C        СЛОВО МОЖЕТ БЫТЬ ДЛИННЕЕ 8 ЛИТЕР,
C        ИГНОРИРОВАТЬ ОСТАТОК
      J = I+8
      DO 225 COL = J, TTLWTH
         IF (TTLCH(COL) .EQ. ' ') GO TO 300
225   CONTINUE
      COL = TTLWTH+1
      GO TO 300
C        СЛОВО ИМЕЕТ ДЛИНУ МЕНЕЕ 8 ЛИТЕР,
C        УСТАНОВИТЬ COL
250   COL = I+J-1
C
C        ПЕРЕСЛАТЬ НАКОПЛЕННЫЕ ЛИТЕРЫ В 'WORD'
300   WORD = WD
      IF (TRACE) WRITE (TRC,350) WORD
350   FORMAT (' *** TRACE *** EXITING NEXTWD, WORD=', A8)
      RETURN
      END

      INTEGER FUNCTION KEYLUK (WORD)
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ЕСТЬ ЛИ 'WORD' В СПИСКЕ КЛЮЧ. СЛОВ. ЕСЛИ НЕТ, ВЕРНУТЬ 0,            *
C* ЕСЛИ ДА—ВЕРНУТЬ ИНДЕКС СООТВЕТСТВУЮЩЕГО КЛЮЧЕВОГО СЛОВА              *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
```

Продолжение Рис. 4.9.

```
      CHARACTER WORD*8
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      IF (TRACE) WRITE (TRC,50) WORD
50    FORMAT (' *** TRACE *** ENTERING KEYLUK, WORD=', A8)
      DO 100 KEYLUK = 1, KEYNUM
         IF (KEYTAB(KEYLUK) .EQ. WORD) GO TO 200
100   CONTINUE
      KEYLUK = 0
200   IF (TRACE) WRITE (TRC,201) KEYLUK
201   FORMAT (' *** TRACE *** EXITING KEYLUK, RETURN VALUE=', I2)
      RETURN
      END

      BLOCK DATA
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИНИЦИАЛИЗАЦИЯ ОБЩИХ ОБЛАСТЕЙ:                                    *
C*   RDR    = 5    :НОМЕР КАНАЛА READER IBM 360-370                 *
C*   PRT    = 6    :НОМЕР КАНАЛА PRINTER IBM 360-370                *
C*   MASTER = 11   :НОМЕР КАНАЛА ДЛЯ ФАЙЛА ЗАГЛАВИЙ                 *
C*   DIRECT = 12   :НОМЕР КАНАЛА ДЛЯ ФАЙЛА С ПРОИЗВОЛЬНЫМ ДОСТУПОМ  *
C*   DBG    = 6    :ОТЛАДОЧНЫЙ ВЫВОД ВМЕСТЕ СО СТАНДАРТНОЙ ПЕЧАТЬЮ  *
C*   TRC    = 6    :ТРАССИРОВКА ВМЕСТЕ СО СТАНДАРТНОЙ ПЕЧАТЬЮ       *
C*   TRACE  = TRUE:ПРОСЛЕЖИВАНИЕ ВСЕХ ВХОДОВ И ВЫХОДОВ ИЗ ПОДПРОГРАММ *
C*   DEBUG  = TRUE:ВКЛЮЧИТЬ ВСЕ ОТЛАДОЧНЫЕ СООБЩЕНИЯ                *
C*   TTLMAX = 100  :РАЗРЕШЕНО НЕ БОЛЕЕ 100 ЗАГЛАВИЙ                 *
C*   KEYMAX = 50   :НЕ БОЛЕЕ 50 КЛЮЧЕВЫХ СЛОВ                       *
C*   TTLWTH = 70   :ДЛИНА НЕ БОЛЕЕ 70 ЛИТЕР                         *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      COMMON/KEY/ TRACE, DEBUG, RDR, PRT, TRC, DBG, MASTER, DIRECT,
     * TTLWTH, TTLNUM, TTLMAX, KEYNUM, KEYMAX, TTLKEY, TITLE, KEYTAB
      INTEGER RDR, PRT, TRC, DBG, MASTER, DIRECT, TTLWTH
      INTEGER TTLNUM, TTLMAX, KEYNUM, KEYMAX
      LOGICAL TRACE, DEBUG, TTLKEY(100,50)
      CHARACTER TITLE*70, KEYTAB*8(50)
      DATA RDR/5/, PRT/6/, MASTER/11/, DIRECT/12/, DBG/6/, TRC/6/
      DATA TRACE/.TRUE./, DEBUG/.TRUE./
      DATA TTLMAX/100/, KEYMAX/50/, TTLWTH/70/
      END
```

Продолжение Рис. 4.9.

Упражнения

4.01. Какой «бит нечетности» следует присоединить к следующим 8-разрядным кодам на 9-дорожечной магнитной ленте?

(а) 10111011
(б) 11111111
(в) 01000000
(г) 00000000

4.02. Предположим, что межблочные промежутки на магнитной ленте составляют 1/2 дюйма, данные записаны с плотностью 800 *bpi* и лента имеет длину 2400 футов.
(а) Если говорят, что 75% ленты используется эффективно, то сколько литер запоминается в отдельном блоке (при условии что все блоки одного размера)?
(б) Если мы предполагаем увеличить коэффициент полезного использования ленты до 90%, какой длины должен быть каждый блок?

4.03. Форматные записи в системе DEC-10 оканчиваются маркером конца записи. Почему этот способ не принят для бесформатных записей?

4.04. Предположим, что в программе для IBM 360-370 выполняется инструкция

DEFINE FILE 12 (100,40,L,I12)

или в программе для DEC-10 — инструкция

CALL DEFINE FILE (12,40,I12,'RAND.DAT')

(а) Чему равно значение целой переменной I12 после выполнения инструкции

READ (12'15,100) DATA

(б) Какое значение имеет переменная I12 после выполнения последовательности инструкций

WRITE (12'6,200) DATA
FIND (12'22)

4.05. Имеет ли смысл выдавать команду REWIND с номером устройства, ассоциированным с последовательным набором данных, хранящимся на устройстве прямого доступа? Объясните ваш ответ.

4.06. Имеет ли смысл выдавать команду REWIND для файлов прямого доступа? Объясните ваш ответ.

4.07. Предположим, что необходимо выбрать устройство для записи последовательного набора данных. Укажите достоинства и недостатки магнитных лент или дисков.

4.08. Рассмотрите DD-команды OS/JCL IBM 360-370:

```
//FT06F001 DD SYSOUT=A
//FT10F001 DD UNIT=TAPE,DSN=NEWDATA,VOL=SER=MYTAPE,LABEL=6,DISP=NEW,
//            DCB=(RECFM=FB,LRECL=84,BLKSIZE=256)
//FT12F001 DD UNIT=TAPE,DISP=NEW,DSN=MODDATA,VOL=SER=OTHER,
//            LABEL=(3,NL),DCB=(RECFM=FB,LRECL=60,BLKSIZE=1200)
//FT14F001 DD UNIT=SYSDA,DSN=DISKDATA,DISP=(NEW,CATLG),
//            SPACE=(200,(20,2)),
//            DCB=(RECFM=F,LRECL=200,BLKSIZE=200)
```

(а) Какой тип устройства связан с номером 6?
(б) Что означает 10 в FT10F001?
(в) Нарисуйте или объясните формат каждой записи в шестом файле ленты с именем MYTAPE.
(г) Нарисуйте формат каждой записи в третьем файле ленты с именем OTHER.
(д) MYTAPE и OTHER используют метки?
(е) Что означают три величины 200, 20 и 2 в параметре SPACE=(200, (20, 2))?
(ж) Напишите DD-команду, которую можно использовать для доступа к вновь созданному набору данных DISKDATA с номером устройства 8 при последующих выполнениях программы.

4.09. Напишите DD-команду, необходимую для создания помеченного файла на магнитной ленте с именем MINE. Магнитная лента уже содержит четыре набора данных. Новый файл располагается за четвертым файлом. Его имя MYDATA. Каждая запись имеет длину 100 литер и сблокирована в физические блоки по 10 записей в блоке. К файлу обращаются по номеру устройства 9.

4.10. Предположим, что выдана следующая последовательность команд:

.MOUNT MTA:14/REELID:MINE/RONLY
.SKIP 14 : 2 FILES

(а) Что означают 14, MINE и RONLY?
(б) Где найти файл?

4.11. Предположим, что требуется записать файл в конце всех файлов, находящихся на магнитной ленте с именем TAPE1. Напишите последовательность

команд в системе DEC-10 так, чтобы для этих целей можно было использовать канал 8.

4.12. Напишите необходимую команду в системе DEC-10, если для обработки набора данных на диске используется канал 9.

4.13. Напишите программу, которая обрабатывает файл на магнитной ленте, содержащей 60-литерные записи. Каждая такая запись состоит из 20-литерного поля имени, за которым расположено 40-литерное поле адреса. Ваша программа вначале должна читать все записи, отмечая каждое отдельное имя и число появлений этого имени. Затем необходимо вновь прочитать файл, печатая последнюю запись, связанную с каждым именем.

4.14. Требуется написать систему заказа авиационных билетов. Входом в программу служит набор перфокарт, каждая из которых имеет следующую структуру:

Позиции:	*Записанная информация*:
1—20	Имя пассажира
21—60	Домашний адрес
61—70	Номер телефона
71	Рейс 1
72	Рейс 2
73	Рейс 3

Перфокарта, завершающая указанный набор перфокарт, содержит в поле имени пассажира 20 звездочек. Система обслуживает не более 100 пассажиров. Каждая входная запись (кроме перфокарты, содержащей 20 звездочек) выводится на дисковый файл с произвольным доступом. Кроме того, следует создать несколько таблиц. Первая таблица NAMES (имена) содержит «А» в элементе NAMES (K), если «А» — имя пассажира K-й записи. Вторая таблица RESERV (заказанные) содержит K в элементе RESERV (I, J) в случае, если пассажир из K-й записи является J-м пассажиром I-го рейса. Можно предположить, что ни один рейс не принимает более 25 пассажиров. Поэтому RESERV объявляется целым массивом размером 9 на 25. Вначале установите в ноль все элементы массива RESERV, чтобы указать, что вначале не было пассажиров ни на один рейс.

Вслед за перфокартами резервирования вы вводите ряд команд.
Каждая перфокарта, содержащая команду, имеет следующую структуру:

Позиции:	*Записанная информация*:
1—4	Код команды
5—24	Имя пассажира (не используется с CANC, PRINT)
25	Номер рейса (не используется с INQ)
26—65	Домашний адрес (не используется с ADD)
66—75	Номер телефона (используется только с ADD)

Ваша программа должна интерпретировать команду и печатать соответствующий вывод для каждой команды.

Разрешены команды:

ADD — Записать пассажира на рейс. Необходимо проверять рейс на переполнение, а пассажиру разрешать резервирование билетов не более чем на три рейса.

DELE — Вычеркнуть пассажира из некоторого рейса. Необходимо следить за попытками вычеркнуть пассажира из неназначенного рейса.

CANC — Отменить резервирование мест для всех пассажиров данного рейса.

PRNT — Напечатать список всех пассажиров на данный рейс.

INQ — Напечатать адрес, номер телефона и номера рейсов, на которые заказано место для данного пассажира.

CONF — Указать, зарезервирован или нет билет данному пассажиру на определенный рейс.

Глава 5

СТРУКТУРЫ ДАННЫХ

Вы уже познакомились со всеми выполняемыми инструкциями языка Фортран. Несмотря на то что число различных типов инструкций невелико, язык тем не менее оказывается полезным при решении широкого круга вычислительных задач. Силу Фортрана определяют не инструкции сами по себе, а то, как они используются при решении задач. Подобно тому как программисты сконструировали синтаксические структуры типа IF-THEN-ELSE или CASE, описанные в гл. 1, были найдены также методы для представления отношений между элементами данных. В этой главе мы рассмотрим некоторые структуры, полезные для представления и обработки данных, в программах на языке высокого уровня.

5.1. Списки

Списки лежат в основе большинства структур данных. *Списком* называется упорядоченный набор элементов данных. Список имеет единственный первый элемент, называемый *головой* списка, а также последний элемент, называемый *хвостом* списка. Каждый элемент списка, кроме головы, имеет единственного предшественника, и каждый элемент, за исключением хвоста, имеет единственного преемника. Материальными примерами списка могут служить телефонный справочник или картотека. Из этих примеров следует, что отдельный элемент списка может состоять из нескольких компонентов; каждая запись в телефонном справочнике, например, содержит фамилию, адрес и номер телефона. Каждый элемент данных называется *записью*, или *узлом*, или *элементом* списка. Компоненты записи называются *полями*.

Над списками обычно допускаются следующие операции.

1. Вставить узел в список.
2. Исключить узел из списка.
3. Соединить два списка в один.
4. Разбить один список на два.
5. Вывести (напечатать) все узлы списка.

6. Найти узел, содержащий определенное значение в некотором поле.
7. Упорядочить узлы списка в соответствии со значениями некоторого поля.

Наиболее распространенной формой списка является *последовательный список*, который можно представить массивом языка Фортран. В последовательном списке каждый элемент физически располагается вслед за своим предшественником. Многие из программ, которые мы рассмотрели в предыдущих главах, выполняют указанные выше операции над последовательными списками. В этой главе мы рассмотрим некоторые разновидности последовательных списков, а также несколько других типов списков.

5.2. Стеки

Представьте себе магазин, где продается множество разнообразных товаров. Когда товар на полках кончается, служащие выкладывают вновь поступившие изделия. Обычно они выставляют новый товар, когда осталось еще немного прежнего товара, поскольку лучше не допускать, чтобы полки были совсем пустыми.

Новые изделия могут иметь другую цену, но служащие выкладывают на полки изделия с разными ценами, полагая, что покупатели сначала раскупят товар по более низкой цене. Если старые товары запылились или помялись, то покупатели в первую очередь возьмут новый товар, пусть даже более дорогой.

Владельцу магазина в любой момент желательно иметь информацию о суммарной стоимости всех товаров, имеющихся в продаже в его магазине. В магазине, где продается много дешевых товаров (например, в бакалейном магазине), вряд ли реально хранить записи, скажем, о том, продана ли банка груш ценой 29 центов, выпущенная 15 июня, или банка груш ценой 31 цент, выпущенная 8 июля. В некоторых случаях это вообще невозможно, поскольку, например, яблоко, сорванное 15 июня, нельзя отличить от яблока, сорванного 18 июня.

Для определения общей стоимости имеющихся товаров в некоторых магазинах применяется метод, называемый LIFO. Аббревиатура LIFO означает «Last In, First Out» (последним пришел — первым ушел), т. е. считается, что товар, поступивший позже, продается в первую очередь. Такой метод учета можно реализовать при помощи структуры данных, называемой стеком.

Стек — это список, в котором включения и исключения производятся только с одного конца. Когда элемент помещается на стек, то элемент, ранее находившийся на вершине стека, стано-

вится временно недоступным. Когда мы удаляем из стека этот новый элемент, то предыдущий элемент, который был недоступен, вновь оказывается на вершине стека. Операции включения и исключения из стека называются соответственно «*положить* на стек» и «*взять* со стека» [1]).

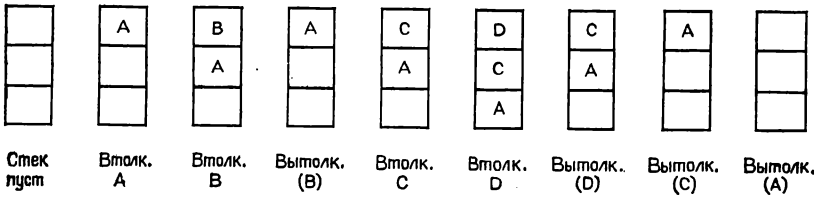


Рис. 5.1. Операции со стеком.

Стопка подносов в кафетерии ведет себя, как стек. Чистые подносы помещаются на вершину стека, и посетители берут подносы сверху. В некоторых кафетериях есть пружинные механизмы для хранения подносов, так что операции «положить» (втолкнуть) и «взять» (вытолкнуть) связаны с физическим продвижением остатка стека. На рис. 5.1 показано несколько операций со стеком. Заметим, что элементы берутся со стека не в том порядке, в котором их туда помещали, а в обратном.

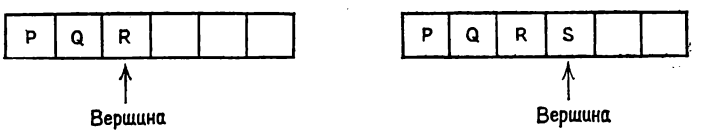


Рис. 5.2. Реализация стека с помощью массива.

При моделировании движения товаров в магазине методом «последним пришел — первым ушел» используется по одному стеку на каждый вид товара. Как только поступает новая партия товара, на стек для этого вида товара помещается элемент, содержащий количество и цену вновь поступившего товара. По мере распродажи товара проданное количество вычитается из верхнего элемента стека; когда количество в верхнем элементе стека становится нулевым, этот элемент удаляется из стека, а остаток вычитается уже из нового верхнего элемента. Общая стоимость товара вычисляется путем перемножения количества товара в каждом элементе стека на его цену и суммирования по всем элементам.

---

[1]) В оригинале «push» (втолкнуть) и «pop» (вытолкнуть). Стек можно представлять себе как пружинный механизм — «магазин»; отсюда и термины «втолкнуть» и «вытолкнуть».— *Прим. перев.*

Стек удобно представлять в Фортране при помощи массива и указателя на верхний элемент. Мы будем изображать стек, считая, что свободное пространство находится не в нижней части массива, как на рис. 5.1, а в верхней. Далее, вместо того чтобы перемещать все данные в стеке всякий раз, когда добавляется новый элемент, мы просто будем продвигать указатель вершины стека. Наконец, мы будем изображать стек «лежащим на боку», так что низ у него окажется слева, а вершина — справа. На рис. 5.2 изображен стек до и после того, как на него был помещен элемент S.

Если стек реализуется массивом фиксированного размера, то возникает неизбежная проблема — нужно следить, чтобы на стек

```
      SUBROUTINE STKOPS (*, STACK, STKSIZ, STKTOP, DATA)
      INTEGER STKSIZ, STKTOP, STACK(STKSIZ), DATA
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C  ПОДПРОГРАММА ОПЕРАЦИЙ СО СТЕКОМ =                                =
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C  ЭТА ПОДПРОГРАММА СОДЕРЖИТ МОДУЛИ ДЛЯ ИНИЦИАЛИЗАЦИИ=
C     СТЕКА И ОПЕРАЦИЙ ПОЛОЖИТЬ НА СТЕК И ВЗЯТЬ СО СТЕКА.          =
C  В КАЖДОЙ ИЗ ЭТИХ ПОДПРОГРАММ СТЕК ПРЕДСТАВЛЕН МАССИВОМ=
C     ТИПА INTEGER;                                                 =
C  УКАЗАТЕЛЬ ТЕКУЩЕЙ ВЕРШИНЫ СТЕКА СОДЕРЖИТСЯ В ПЕРЕМЕННОЙ=
C     STKTOP, РАЗМЕР (DIMENSION) СТЕКА РАВЕН STKSIZ                =
C  ПРОГРАММИСТ: — — —                ДАТА: — — —                   =
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C
      ENTRY STKPSH (*, STACK, STKSIZ, STKTOP, DATA)
C — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —
C  ЭТОТ ВХОД СЛУЖИТ ДЛЯ ПОМЕЩЕНИЯ ЗНАЧЕНИЯ ПЕРЕМЕННОЙ "DATA"—
C  НА ВЕРШИНУ СТЕКА (МАССИВ "STACK").                             —
C  ЕСЛИ СТЕК ПЕРЕПОЛНИЛСЯ, ТО АЛЬТЕРНАТИВНЫЙ ВОЗВРАТ RETURN 1.—
C  ПРИ УСПЕШНОМ ВЫПОЛНЕНИИ ОПЕРАЦИИ STKTOP БУДЕТ УКАЗЫВАТЬ—
C  НА НОВУЮ ВЕРШИНУ СТЕКА.                                        —
C  — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —
C
C . . . . . . ПРОВЕРКА НА ПЕРЕПОЛНЕНИЕ СТЕКА
      IF (STKTOP .GE. STKSIZ)  RETURN 1
C . . . . . . ПЕРЕПОЛНЕНИЯ НЕТ, ПОМЕСТИТЬ ЭЛЕМЕНТ НА СТЕК
      STKTOP = STKTOP+1
      STACK(STKTOP) = DATA
      RETURN
C
C
      ENTRY STKPOP (*, STACK, STKSIZ, STKTOP, DATA)
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C  ЭТОТ ВХОД СЛУЖИТ ДЛЯ УДАЛЕНИЯ ТЕКУЩЕГО ВЕРХНЕГО ЭЛЕМЕНТА СТЕКА  -
C  И ПРИСВАИВАНИЯ ЕГО ЗНАЧЕНИЯ ПЕРЕМЕННОЙ „DATA".                  -
C  ЕСЛИ НЕХВАТКА В СТЕКЕ, ТО АЛЬТЕРНАТИВНЫЙ ВЫХОД RETURN 1.        -
C  ПРИ УСПЕШНОМ ВЫПОЛНЕНИИ ОПЕРАЦИИ STKTOP БУДЕТ УКАЗЫВАТЬ         -
C  НА НОВУЮ ВЕРШИНУ СТЕКА.                                          -
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C
C . . . . . . ПРОВЕРКА НА НЕХВАТКУ В СТЕКЕ
      IF (STKTOP .LE. 0)  RETURN 1
C . . . . . . НЕХВАТКИ НЕТ, ВЗЯТЬ ЭЛЕМЕНТ СО СТЕКА
      DATA = STACK(STKTOP)
      STKTOP = STKTOP-1
      RETURN
```

Рис. 5.3. Подпрограмма операций со стеком.

```
C
C
      ENTRY STKINT (*, STACK, STKSIZ, STKTOP, DATA)
C-----------------------------------------------------------
C  ЭТОТ ВХОД—ДЛЯ ИНИЦИАЛИЗАЦИИ СТЕКА.                       -
C  УКАЗАТЕЛЬ ВЕРШИНЫ И ВЕСЬ СТЕК ЗАПОЛНЯЮТСЯ НУЛЯМИ.         -
C  ВЫХОД RETURN 1 И ПАРАМЕТР „DATA" НЕ ИСПОЛЬЗУЮТСЯ.         -
C-----------------------------------------------------------
      STKTOP = 0
      DO 10 I = 1, STKSIZ
         STACK(I) = 0
   10 CONTINUE
      RETURN
      END
```

Продолжение Рис. 5.3.

не помещали элементов больше, чем может уместиться в массиве. Такая ситуация называется *переполнением* стека. Противоположная ситуация — *недостаток* в стеке — возникает, когда пытаются взять элемент из пустого стека. В программах для работы со стеком необходимо проверять оба эти условия.

Рассмотрим теперь подпрограмму (STKOPS, рис. 5.3), в которой реализованы операции со стеком; каждой операции в подпрограмме соответствует точка входа. Операции «положить на стек» (втолкнуть) и «взять со стека» (вытолкнуть) реализованы в точности так, как описано выше. Точками входа для них являются STKPSH и STKPOP.

Еще одна точка входа (STKINT) нужна для инициализации стека. Поскольку указатель вершины стека должен ссылаться на текущий верхний элемент, то пустой стек будет представлен значением STKTOP=0. Нет необходимости инициализировать сам стек, так как при каждой операции «положить» изменяется содержимое ячейки STACK(STKTOP+1). Для отладки, однако, полезно распечатывать содержимое всего стека или первые несколько его элементов и уметь отличать элементы стека от неинициализированных позиций в нем. По этой причине в программе инициализации заполняются нулями переменная STKTOP и весь стек.

Теперь вы, несомненно, сумеете написать программу, которая бы читала карты, содержащие количество поступившего товара и его цену или количество проданного товара, и определяла суммарную стоимость товара путем просмотра стека.

Стеки являются примером *динамических* структур данных; это означает, что структура во время выполнения программы логически растет или сокращается. Поскольку эта структура реализуется в Фортране при помощи простых переменных и массивов, то она фактически не меняет своего размера, так как переменные и массивы сохраняют постоянный размер. Меняется же размер используемой в каждый момент времени части массива. Ограничения, связанные с реализацией стека переменного разме-

ра при помощи массива фиксированного размера, обычно оказываются не слишком обременительными, поскольку в большинстве приложений можно разумно оценить максимальный возможный размер стека.

5.3. Очереди

Система «последним пришел — первым ушел» для учета товаров не реалистична. Владельцы магазинов не захотят задерживать у себя большое количество старых товаров. Обычно, раскладывая на полках товары, они помещают более новые изделия позади старых, чтобы товары не успели слишком состариться, прежде чем их продадут. Это особенно важно для скоропортящихся или сезонных товаров. В подобных случаях более подходящей оказывается система учета FIFO. В такой системе предполагается, что всякий раз продается наиболее старое из имеющихся изделие данного типа. Сокращение FIFO означает "First In, First Out" (первым пришел — первым ушел), т. е. удаляется всегда самый старый элемент. Такая система моделируется очередью.

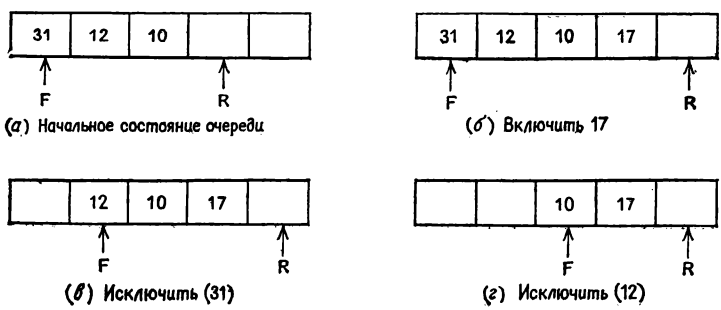
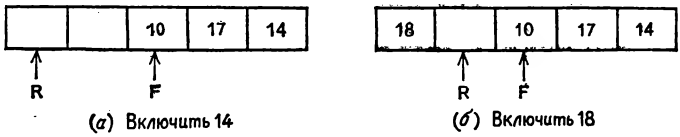


Рис. 5.4. Операции с очередью.

Очередь — это структура данных, в которой элементы добавляются всегда с одного края, а удаляются с другого. Край, в котором выполняются включения, называется *концом* очереди, а край, в котором производятся исключения элементов,— *началом* очереди. Вы знакомы, например, с очередью людей у кассы в магазине самообслуживания или очередью автомобилей у бензозаправочной станции. Вновь прибывшие становятся в один край очереди и покидают ее после оплаты покупок или заправки с другого.

Так же как стек, очередь может быть представлена с помощью массива. Нужны два указателя — на начало и на конец очереди. На рис. 5.4 показана очередь, растущая вправо; начало у нее

слева, а конец справа. Изображено выполнение четырех операций с очередью.



(а) Включить 14 (б) Включить 18

Рис. 5.5. Циклическое представление очереди.

```
      SUBROUTINE QUEOPS (*, QUEUE, QUESIZ, QUEF, QUER, DATA)
      INTEGER QUESIZ, QUEF, QUER, QUEUE(QUESIZ), DATA
      INTEGER TRUER
      DATA IOUT/6/
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C  ПОДПРОГРАММА ДЛЯ РАБОТЫ С ОЧЕРЕДЯМИ =                              *
C* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C  НАБОР МОДУЛЕЙ ДЛЯ РАБОТЫ С ПОСЛЕДОВАТЕЛЬНЫМИ ОЧЕРЕДЯМИ.          *
C  ОЧЕРЕДЬ ХРАНИТСЯ В МАССИВЕ„QUEUE“РАЗМЕРА QUESIZ.                   *
C  МАССИВ ЦИКЛИЧЕСКИЙ: ЗА ПОЗИЦИЕЙ„QUESIZ“                            *
C  СЛЕДУЕТ ПОЗИЦИЯ„1“.                                                *
C  УКАЗАТЕЛИ НА НАЧАЛО И КОНЕЦ ОЧЕРЕДИ НАХОДЯТСЯ                      *
C  В ПЕРЕМЕННЫХ QUEF И QUER.                                          *
C  „DATA“ СОДЕРЖИТ ВКЛЮЧАЕМЫЙ И ИСКЛЮЧАЕМЫЙ ЭЛЕМЕНТ                  *
C  ПРОГРАММИСТ: ---   ДАТА: ---                                       *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C
      ENTRY QUEINS (*, QUEUE, QUESIZ, QUEF, QUER, DATA)
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C  В ЭТОМ МОДУЛЕ ЭЛЕМЕНТ „DATA“ ВКЛЮЧАЕТСЯ В ОЧЕРЕДЬ „QUEUE“.         -
C  ЕСЛИ ВКЛЮЧЕНИЕ УСПЕШНО, ТО УКАЗАТЕЛИ НА НАЧАЛО И КОНЕЦ,            -
C  QUEF И QUER, ИЗМЕНЯЮТСЯ.                                            -
C  ЕСЛИ ПЕРЕПОЛНЕНИЕ, ТО ВОЗВРАТ RETURN 1,                            -
C  QUEF И QUER НЕ МЕНЯЮТСЯ.                                            -
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C
C . . . . . . ПРОВЕРКА НА ПЕРЕПОЛНЕНИЕ
      IF (QUEF .EQ. QUER+1 .OR.
     X    (QUEF .EQ. 1 .AND. QUER .EQ. QUESIZ) )  RETURN 1
C . . . . . . ПЕРЕПОЛНЕНИЯ НЕТ, ВКЛЮЧИТЬ ЭЛЕМЕНТ
      QUEUE(QUER) = DATA
      QUER = QUER + 1
      IF (QUER .GT. QUESIZ) QUER = 1
      RETURN
C
C
      ENTRY QUEDEL (*, QUEUE, QUESIZ, QUEF, QUER, DATA)
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C  ИСКЛЮЧЕНИЕ ИЗ ОЧЕРЕДИ.                                              -
C  В ЭТОМ МОДУЛЕ ЭЛЕМЕНТ ИСКЛЮЧАЕТСЯ ИЗ НАЧАЛА ОЧЕРЕДИ „QUEUE“,         -
C  А ЕГО ЗНАЧЕНИЕ ПРИСВАИВАЕТСЯ ПЕРЕМЕННОЙ „DATA“.                     -
C  ЕСЛИ ОЧЕРЕДЬ ПУСТА, ТО ВОЗВРАТ RETURN 1,                            -
C  QUEF И QUER НЕ МЕНЯЮТСЯ.                                            -
C  ЕСЛИ ИСКЛЮЧЕНИЕ УСПЕШНО, ТО QUEF И QUER ИЗМЕНЯЮТСЯ.                 -
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C
C . . . . . . . . . ПРОВЕРИТЬ, НЕ ПУСТА ЛИ ОЧЕРЕДЬ
      IF (QUEF .EQ. QUER)  RETURN 1
C . . . . . . . . . ОЧЕРЕДЬ НЕ ПУСТА, ИСКЛЮЧИТЬ ЭЛЕМЕНТ
      DATA = QUEUE(QUEF)
      QUEF = QUEF + 1
      IF (QUEF .GT. QUESIZ) QUEF = 1
      RETURN
C
C
```

Рис. 5.6. Операции с очередью. (Текст в инстр. 21:('ОЧЕРЕДЬ: УКАЗАТЕЛЬ НАЧАЛА=', I5, ' УКАЗАТЕЛЬ КОНЦА=' I5/'ЭЛЕМЕНТЫ ПЕЧАТАЮТСЯ В ПОРЯДКЕ ОЧЕРЕДИ'))

```
      ENTRY QUEINT (*, QUEUE, QUESIZ, QUEF, QUER, DATA)
C-----------------------------------------------------------
C МОДУЛЬ, ИНИЦИАЛИЗИРУЮЩИЙ ВКЛЮЧЕНИЯ                        -
C МАССИВ QUEUE ЗАПОЛНЯЕТСЯ НУЛЯМИ,                          -
C А QUEF И QUER БУДУТ УКАЗЫВАТЬ НА ПЕРВЫЙ ЭЛЕМЕНТ.         -
C QUEF.EQ.QUER ПРИЗНАК ПУСТОЙ ОЧЕРЕДИ, ПОЭТОМУ ПРИ ПОПЫТКЕ  -
C ВЫПОЛНИТЬ ИСКЛЮЧЕНИЕ РАНЬШЕ ВКЛЮЧЕНИЯ                     -
C ПРОИЗОЙДЕТ ВЫХОД „ПО НЕХВАТКЕ“.                          -
C В ЭТОМ МОДУЛЕ НЕ ИСПОЛЬЗУЕТСЯ ВЫХОД RETURN 1 И „DATA“     -
C-----------------------------------------------------------

      QUEF = 1
      QUER = 1
      DO 10 I = 1, QUESIZ
         QUEUE(I) = 0
   10 CONTINUE
      RETURN
C
C
      ENTRY QUEDMP (*, QUEUE, QUESIZ, QUEF, QUER, DATA)
C-----------------------------------------------------------
C РАСПЕЧАТЫВАЕТСЯ МАССИВ „QUEUE“ В ПОРЯДКЕ                  -
C ОЧЕРЕДИ И УКАЗАТЕЛИ НА НАЧАЛО (QUEF) И КОНЕЦ              -
C (QUER) ОЧЕРЕДИ.                                          -
C ВЫХОД RETURN 1 И „DATA“ НЕ ИСПОЛЬЗУЮТСЯ.                 -
C-----------------------------------------------------------
      WRITE (IOUT, 21) QUEF, QUER
      IF (QUEF .EQ. QUER)  RETURN
C . . . . . . . . . ВЫЧИСЛИТЬ УКАЗАТЕЛЬ НА ИСТИННЫЙ КОНЕЦ
      TRUER = QUER-1
      IF (TRUER .EQ. 0) TRUER = QUESIZ
C . . . . . . . . . ПОДРЯД ИЛИ ЧЕРЕЗ КОНЕЦ МАССИВА?
      IF (QUEF .GT. TRUER) GO TO 100
C . . . . . . . . . ПОДРЯД
      WRITE (IOUT,22) (QUEUE(I), I=QUEF, TRUER)
      RETURN
C . . . . . . . . . ОЧЕРЕДЬ РАСПАДАЕТСЯ НА ДВЕ ЧАСТИ
  100 WRITE (IOUT,22) (QUEUE(I), I=QUEF,QUESIZ), (QUEUE(I), I=I,TRUER)
      RETURN
   21 FORMAT ('0QUEUE DUMP:  FRONT POINTER = ', I5, '  REAR POINTER =',
     #        I5/  ' ANY ELEMENTS FOLLOW, IN ORDER FROM FRONT TO REAR')
   22 FORMAT (1X,10I8)
      END
```

Продолжение Рис. 5.6.

При реализации стека в виде массива его низ был фиксирован, а вершина могла изменять свое положение. В очереди же могут двигаться как начало, так и конец, поэтому активная часть очереди постоянно перемещается вдоль массива. Если добавить еще один элемент в очередь на рис. 5.4, *г*, то он займет последнюю ячейку массива, и очередь окажется вроде бы заполненной. Имеются, однако, две свободные позиции в левом конце массива. Эти позиции можно использовать, если рассматривать массив как циклическую структуру, соединив два его конца. Позицией, правой по отношению к крайней правой позиции массива, будет крайняя левая позиция массива. Две вставки в такой массив изображены на рис. 5.5.

Так же как и для стека, возможны ситуации переполнения и нехватки; их необходимо выявлять в программах, работающих с очередями. На рис. 5.6 дана программа с входными точками для инициализации, вставок и удалений из очередей, имеющих структуру, как на рис. 5.4 и 5.5. Указатель начала очереди указывает

всегда на самый старый элемент в очереди; указатель конца очереди всегда указывает на одну позицию правее самого нового элемента. Такое на первый взгляд неестественное представление указателя конца очереди оказывается удобным, когда требуется отличить пустой список (R=F) от полного (R=F—1). При данной интерпретации указателя конца очереди в массиве размера n, используемом для представления очереди, может храниться лишь (n—1) элементов.

В дополнение к стандартному набору операций с очередью наша подпрограмма содержит еще одну входную точку (QUEDMP), полезную для отладки. Одно из самых неприятных занятий, связанных с отладкой программы,— тратить время на получение разумных отладочных выдач. Потратив же несколько минут на написание этих нескольких инструкций в момент написания программ, вы можете ускорить процесс отладки.

Мы снова воспользовались для представления динамической структуры данных массивом фиксированного размера. Хотя такой метод представления и прост, но с ним, как мы увидим в следующем разделе, связана проблема использования памяти.

5.4. Последовательное распределение памяти с указателями

В связи с последовательными распределениями памяти возникает несколько проблем. Одна из них связана с последовательными списками общего типа, не являющимися стеками или очередями. Предположим, что список отсортирован в возрастающем порядке, а затем понадобилось исключить несколько элементов из середины списка. После каждого исключения нужно упаковывать список, т. е. подвигать все следующие за исключенным элементы на одну позицию вверх. Это необходимо для поддержания последовательной структуры списка. В некоторых процессах, например в алгоритме бинарного поиска из гл. 1, предполагается, что элементы списка расположены подряд, без пропусков или пустых позиций между двумя соседними элементами. Но перемещение данных, вызванное упаковкой списка, может отнять слишком много времени, если список длинный, а исключаемый элемент находится близко к началу.

Кроме того, узел списка может содержать много полей. Нетрудно представить список таких элементов в виде ряда параллельных массивов, где каждый массив содержит определенное поле всех записей; j-я запись представляется j-ми элементами во всех массивах. Если мы сортируем записи такого списка по некоторому полю и метод сортировки требует взаимных перестановок элементов, то всякий раз, когда необходимо поменять местами записи, приходится переставлять элементы во всех массивах.

Такой процесс перемещения данных сразу во всех массивах может оказаться слишком медленным.

Эту проблему можно разрешить при помощи дополнительного массива NEXT, который освободит нас от ограничений, связанных с последовательными списками. Не обязательно физически размещать элементы в памяти подряд, можно располагать их в

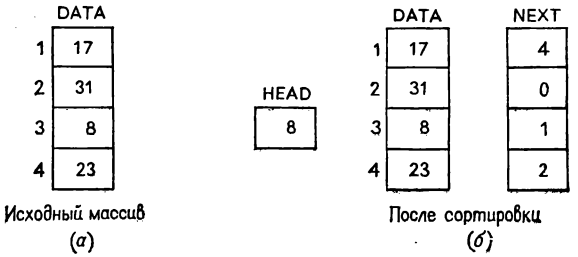


Рис. 5.7. Сортировка с использованием массива NEXT.

произвольных позициях, но для I-го элемента поле NEXT (I) будет содержать индекс элемента I+1. Одна простая переменная HEAD будет содержать индекс первого элемента списка. Поле NEXT(HEAD) будет указывать на второй элемент, индекс третьего элемента содержится в поле NEXT (NEXT(HEAD)) и т. д. При сортировке списков такого типа меняются местами лишь элементы массива NEXT и отпадает необходимость перемещать большое количество данных. Этот метод проиллюстрирован на рис. 5.7.

Обратите внимание, что данные массива DATA в процессе сортировки не перемещались. А это значит, что, будь у нас хоть десять массивов с данными, их элементы также не перемещались бы во время сортировки. Сортировка с использованием массива NEXT ненамного сложнее обычной. В подпрограмме на рис. 5.8 проиллюстрировано развитие этих идей. Предполагается, что в массиве размером NR*NC каждый из NC столбцов содержит однородные данные. Данные в столбцах сортируются так, чтобы обеспечивался возрастающий порядок в столбце SORTC.

В подпрограмме применяется метод пузырька, поэтому время сортировки будет порядка NR**2. Используя массив NEXT, мы вынуждены двигаться по списку только последовательно; поэтому нельзя воспользоваться лучшим алгоритмом поиска, так как для этого требуется уметь просматривать список в другом порядке, а не только последовательно сверху вниз. В упражнениях, однако, рассматривается метод с применением массива указателей, который позволяет воспользоваться более эффективными алгоритмами сортировки.

```
      SUBROUTINE INSORT (A, NR, NC, COL, NEXT, HEAD)
      INTEGER A(NR,NC), NEXT(NR), HEAD, COL
      INTEGER NLESS2, P, K, L, I, J
      LOGICAL SWAP
C========= ТАКАЯ СОРТИРОВКА УМЕНЬШАЕТ ПЕРЕМЕЩЕНИЕ ДАННЫХ ============
C ПРОИЗВОДИТСЯ СОРТИРОВКА НА МЕСТЕ МАССИВА „A" ПО СТОЛБЦУ „COL",      =
C НА ВЫХОДЕ „HEAD" — НОМЕР СТРОКИ С НАИМЕНЬШИМ ЗНАЧЕНИЕМ „COL",       =
C NEXT(HEAD)— НОМЕР ВТОРОЙ СТРОКИ,                                    =
C NEXT(NEXT(HEAD))— НОМЕР ТРЕТЬЕЙ, И Т.Д.                             =
C ПРОГРАММИСТ: ---                          ДАТА: ---                 =
C=====================================================================
C                                                                     =
      HEAD = 1
      IF (NR.LE.1) GO TO 20
      NLESS2 = NR-2
C. . . . . . . . . НАЙТИ НАИМЕНЬШИЙ ЭЛЕМЕНТ, УСТАНОВИТЬ „HEAD"
C. . . . . . . . . ИНИЦИАЛИЗИРОВАТЬ МАССИВ „NEXT"
      DO 10 I=2,NR
         NEXT(I-1) = I
         IF (A(I,COL) .LT. A(HEAD,COL)) HEAD = I
   10 CONTINUE
C. . . . . . . . . „HEAD" УКАЗЫВАЕТ НА НАИМЕНЬШИЙ ЭЛЕМЕНТ
C. . . . . . . . . СДЕЛАТЬ ЭЛЕМЕНТ (HEAD-1) КОНЦОМ СПИСКА
      IF (HEAD.EQ.1) GO TO 20
         NEXT(HEAD-1) = 0
         NEXT(NR) = 1
      GO TO 30
   20 NEXT(NR) = 0
C. . . . . . . . . ВЫПОЛНИТЬ ОБЫЧНУЮ СОРТИРОВКУ МЕТОДОМ ПУЗЫРЬКА,
C. . . . . . . . . ЗАМЕНИВ ВЕЗДЕ „A" НА „NEXT"
   30 IF (NR.LE.2) RETURN
      DO 60 I = 1,NLESS2
         SWAP = .FALSE.
         P = HEAD
         DO 50 J = I,NLESS2
            K = NEXT(P)
            L = NEXT(K)
C. . . . . . . . .          ------      ------      ------
C. . . . . . . . .  ---    |      | =>|      | =>|      |
C. . . . . . . . .          ------      ------      ------
C. . . . . . . . .          УЗЕЛ P      УЗЕЛ K      УЗЕЛ L
C. . . . . . . . . ЕСЛИ K И L НЕУПОРЯДОЧЕНЫ, ПОМЕНЯТЬ ИХ МЕСТАМИ
C. . . . . . . . .
            IF (A(K,COL) .LE. A(L,COL)) GO TO 40
               NEXT(P) = L
               NEXT(K) = NEXT(L)
               NEXT(L) = K
               P = L
               SWAP = .TRUE.
               GO TO 50
   40       P = K
   50    CONTINUE
         IF (.NOT. SWAP) RETURN
   60 CONTINUE
      RETURN
      END
```

Рис. 5.8. Сортировка на том же месте.

5.5. Связанное распределение

Добавив поле NEXT, мы получаем дополнительное преимущество — один и тот же блок памяти можно использовать для элементов разных списков. Вспомните пример с учетом товаров. Там для каждого вида товаров требовался стек или очередь; если в магазине продается 10 000 видов изделий, то нужно 10 000 стеков или очередей. Предположим, что один из товаров поступил в магазин по 50 различным заказам, в то время как в среднем каждый товар поступает в магазин по 2—3 заказам. Если невоз-

можно заранее предсказать, в какой очереди будет 50 элементов, а в какой — всего несколько, то под каждую из 10 000 очередей нужно будет отвести место на 50 элементов, т. е. 500 000 ячеек. Из этих 500 000 ячеек, однако, в среднем 470 000 использоваться не будут. Естественно, такой ситуации хотелось бы избежать.

Добавим к каждой записи одно поле; это дополнительное поле будет использоваться так же, как NEXT в предыдущем разделе. Назовем его полем LINK и будем хранить в нем адрес следующего элемента списка. Такой список называется *связанным списком*, и говорят, что каждый узел *связан со* следующим или

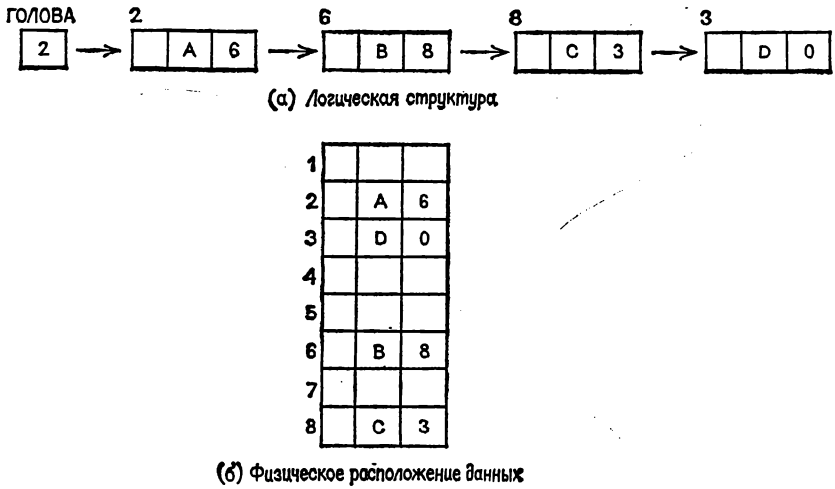


Рис. 5.9. Представление связанных списков.

указывает на следующий узел. С логической точки зрения наша структура данных отличается от своего физического представления. Логическая структура и ее физическое представление изображены на рис. 5.9.

В связанном списке на рис. 5.9, *б* имеются незаполненные позиции, но они не являются частью связанного списка, изображенного на рис. 5.9, *а*; связанный список состоит всего из четырех элементов. Пустые ячейки можно было бы отвести под элементы другого списка. Связанное распределение позволяет формировать два или более списков в одной и той же общей области памяти.

На рис. 5.9, *б*, например, из элементов 1, 4, 5 и 7 можно было бы составить другой список. Для нового списка понадобился бы указатель на начало, скажем HEAD2. HEAD2 указывал бы на первый элемент второго списка; пусть это будет элемент 4. Указатель LINK(4) ссылался бы на второй элемент, допустим эле-

мент 7; в LINK(7) хранилась бы ссылка на третий элемент, скажем элемент 1; наконец, в LINK(1) была бы ссылка на по-

```
      SUBROUTINE LSTKOP (*, STKTOP, PTR, STKDBG)
      INTEGER STKDTA, STKLNK, PTR, STKTOP
      LOGICAL STKDBG
      COMMON STKDTA(1000), STKLNK(1000)
      DATA IOUT/6/

C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C ПОДПРОГРАММЫ ДЛЯ РАБОТЫ СО СВЯЗАННЫМ СТЕКОМ=                       =
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C                                                                     =
C В ЭТИХ ПОДПРОГРАММАХ ВЫПОЛНЯЮТСЯ ОПЕРАЦИИ "ПОЛОЖИТЬ"=
C    И "ВЗЯТЬ"                                                        =
C ДЛЯ СТЕКОВ, ПРЕДСТАВЛЕННЫХ В ВИДЕ СВЯЗАННЫХ СПИСКОВ.               =
C КАЖДЫЙ УЗЕЛ СТЕКА СОСТОИТ ИЗ ДВУХ ПОЛЕЙ:                            =
C    ПОЛЯ ДАННЫХ "STKDTA" И ПОЛЯ СВЯЗИ "STKLNK".                     =
C УКАЗАТЕЛЬ НА ВЕРШИНУ СТЕКА ХРАНИТСЯ В "STKTOP".                     =
C МАССИВЫ "STKDTA" И "STKLNK" ХРАНЯТСЯ В COMMON-БЛОКЕ.                =
C "PTR" — ЭТО УКАЗАТЕЛЬ НА ВКЛЮЧАЕМЫЙ ИЛИ ИСКЛЮЧАЕМЫЙ ИЗ=
C    СТЕКА УЗЕЛ.                                                      =
C АЛЬТЕРНАТИВНЫЙ ВОЗВРАТ RETURN 1 ИСПОЛЬЗУЕТСЯ ПРИ ПЕРЕ-=
C ПОЛНЕНИИ ИЛИ НЕХВАТКЕ. ВКЛЮЧАЕМЫЕ И ИСКЛЮЧАЕМЫЕ УЗЛЫ=
C    ПЕЧАТАЮТСЯ, ЕСЛИ ПРИЗНАК ОТЛАДКИ "STKDBG" РАВЕН TRUE.          =
C ПРОГРАММИСТ: — — —                    ДАТА: — — —                   =
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C
C
      ENTRY LSTPOP (*, STKTOP, PTR, STKDBG)
C — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —
C В ЭТОМ МОДУЛЕ ИСКЛЮЧАЕТСЯ ЭЛЕМЕНТ ИЗ СВЯЗАННОГО СТЕКА.          —
C ЕСЛИ ИСКЛЮЧЕНИЕ ПРОШЛО УСПЕШНО, ТО УКАЗАТЕЛЬ НА ИСКЛЮ-—
C ЧЕННЫЙ ЭЛЕМЕНТ НАХОДИТСЯ В "PTR". ЕСЛИ НЕХВАТКА, ТО АЛЬ-—
C ТЕРНАТИВНЫЙ ВЫХОД RETURN 1                                         —
C — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

      IF (STKTOP .LE. 0) RETURN 1
      PTR = STKTOP
      STKTOP = STKLNK(STKTOP)
      IF (STKDBG) WRITE (IOUT,11) PTR, STKDTA(PTR), STKTOP
   11 FORMAT (1X, 'STACK POP: NODE #',I5,' POPPED. DATA IS ',
     #   I6,',  NEW TOP IS ', I5)
      RETURN

C
C
      ENTRY LSTPSH (*, STKTOP, PTR, STKDBG)
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C В ЭТОМ МОДУЛЕ ЭЛЕМЕНТ ПОМЕЩАЕТСЯ НА СТЕК, ПРЕДСТАВЛЕННЫЙ          -
C В ВИДЕ СВЯЗАННОГО СПИСКА.                                          -
C УКАЗАТЕЛЬ НА ВКЛЮЧАЕМЫЙ УЗЕЛ ХРАНИТСЯ В PTR.                       -
C ЕСЛИ УКАЗАТЕЛЬ PTR НЕПРАВИЛЬНЫЙ, ТО ВОЗВРАТ RETURN 1.              -
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      IF (PTR.LE.0) RETURN 1
      IF (STKDBG) WRITE (IOUT, 12) STKDTA(PTR), STKTOP, PTR
   12 FORMAT(1X, 'STACK PUSH. DATA PUSHED IS ', I6, ',  OLD TOP IS ',
     #   I5, ',  NEW TOP IS ', I5)
      STKLNK(PTR) = STKTOP
      STKTOP = PTR
      RETURN
      END
```

Рис. 5.10. Операции со связанным стеком. (Текст в инстр. 11: (1X, ' ВЗЯТЬ СО СТЕКА: УЗЕЛ НОМЕР', I5, 'ДАННЫЕ' I6, ', НОВАЯ ВЕРШИНА', I5); 12: (1X, 'ПОЛОЖИТЬ НА СТЕК: ЭЛЕМЕНТ', I6, СТАРАЯ ВЕРШИНА, I5, ', НОВАЯ ВЕРШИНА', I5))

следний элемент второго списка, элемент 5. Таким образом, два или более списков могут располагаться в общей области памяти.

Вспомним теперь наши подсчеты, согласно которым под стеки, содержащие в сумме 30 000 элементов, понадобилось бы 500 000 ячеек. Применив описанный выше метод, можно сократить требуемый объем памяти до 30 000 позиций на 30 000 элементов. На рис. 5.10 приведена программа для выполнения включений и исключений из связанного стека.

Эта подпрограмма очень похожа на подпрограмму для работы с обычным стеком из разд. 5.2. Тут нет ничего удивительного, поскольку выполняются по существу те же самые действия, а изменилась только структура данных. Поскольку стек является наиболее простой для реализации динамической структурой данных, то он часто используется для хранения свободных узлов памяти, которые можно присоединить к любому списку. Такой список свободных узлов часто называют *списком свободного пространства* или *LAVS* (list of available space).

В разд. 5.3 для решения задачи о моделировании потока товаров в магазине мы ввели очереди, или списки типа FIFO. Для представления связанного стека нужен указатель на голову

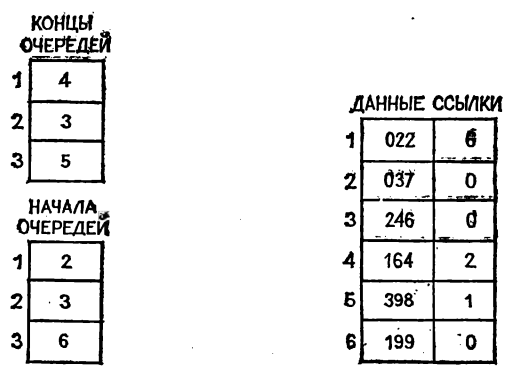


Рис. 5.11. Массивы начал и концов списков.

списка и один или более узлов данных, каждый из которых содержит поле связи. Аналогичным образом представляются связанные очереди: для каждой очереди нужен один указатель начала, один указатель конца и один или более узлов данных со связями. Пусть в магазине продается 10 000 видов товаров, тогда потребуется 10 000 указателей начала очередей и 10 000 указателей их концов; указатели можно представить в виде двух массивов. Каждый из этих массивов является списком, и каждый элемент этих списков указывает на другой список. Массивы такого

вида образуют *список списков*, пример которого изображен на рис. 5.11.

В этом примере первый список состоит из двух элементов, ячейки 4 и 2; список 2 состоит из единственного элемента, ячейки 3; список 3 содержит три элемента, ячейки 5, 1 и 6.

Работа со списком списков происходит, по существу, так же, как с одним списком. Программа для работы с очередями получает в качестве параметров указатели начала и конца очереди, и ее не интересует, являются ли эти указатели простыми переменными или элементами массивов. На рис. 5.12 даны программы для работы с очередью, представленной в виде связанного списка.

```
      SUBROUTINE LQUEOP (*, QUEF, QUER, PTR)
      INTEGER QUEF, QUER, PTR, QUEDTA, QUELNK
      COMMON QUEDTA(1000), QUELNK(1000)

C  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C  ПОДПРОГРАММЫ ДЛЯ РАБОТЫ СО СВЯЗАННОЙ ОЧЕРЕДЬЮ                 =
C  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C  ЭТИ ПОДПРОГРАММЫ ВЫПОЛНЯЮТ ВКЛЮЧЕНИЕ И ИСКЛЮЧЕНИЕ ИЗ=
C  СВЯЗАННЫХ ОЧЕРЕДЕЙ. УКАЗАТЕЛИ НАЧАЛА И КОНЦА ОЧЕРЕДИ=
C     ХРАНЯТСЯ СООТВЕТСТВЕННО В "QUEF" И "QUER".                =
C  УКАЗАТЕЛЬ НА ВКЛЮЧАЕМЫЙ ИЛИ ИСКЛЮЧАЕМЫЙ УЗЕЛ НАХО-=
C     ДИТСЯ В "PTR".                                            =
C  УЗЛЫ ОЧЕРЕДИ СОСТОЯТ ИЗ ДВУХ ПОЛЕЙ И ХРАНЯТСЯ В МАССИВАХ=
C     "QUEDTA" И "QUELNK".                                      =
C  ПРИ НЕХВАТКЕ — АЛЬТЕРНАТИВНЫЙ ВЫХОД RETURN 1.               =
C  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C
C
      ENTRY QUEADD (*, QUEF, QUER, PTR)
C  — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —
C  В ЭТОЙ ПОДПРОГРАММЕ В ОЧЕРЕДЬ, ПРЕДСТАВЛЕННУЮ В ВИДЕ СВЯ-—
C  ЗАННОГО СПИСКА, ДОБАВЛЯЕТСЯ УЗЕЛ. УКАЗАТЕЛЬ НА НОВЫЙ УЗЕЛ—
C  НАХОДИТСЯ В "PTR".                                          —
C  — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

C. . . . . УБЕДИТЬСЯ В СУЩЕСТВОВАНИИ ДОБАВЛЯЕМОГО УЗЛА
      IF (PTR .LE. 0)  RETURN 1
C. . . . . . . . ДОБАВИТЬ УЗЕЛ В КОНЕЦ, ЕСЛИ ОЧЕРЕДЬ НЕ ПУСТА
      IF (QUER .NE. 0)  QUELNK(QUER) = PTR
      QUELNK(PTR) = 0
      QUER = PTR
C. . . . . . . . ЕСЛИ СПИСОК БЫЛ ПУСТ, УСТАНОВИТЬ УКАЗАТЕЛЬ НАЧАЛА
      IF (QUEF.EQ.0) QUEF = PTR
      RETURN
C
C
      ENTRY QUEDEL (*, QUEF, QUER, PTR)
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C  ИСКЛЮЧЕНИЕ ИЗ СВЯЗАННОЙ ОЧЕРЕДИ; ИСКЛЮЧАЕТСЯ ПЕРВЫЙ ЭЛЕМЕНТ,  -
C  А УКАЗАТЕЛЬ НА НЕГО ПОМЕЩАЕТСЯ В „PTR“                         -
C  ЕСЛИ ОЧЕРЕДЬ ПУСТА, ТО ВОЗВРАТ RETURN 1.                      -
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      IF (QUEF .EQ. 0)  RETURN 1
C. . . . . . . НАЙТИ ПЕРВЫЙ ЭЛЕМЕНТ, ИСКЛЮЧИТЬ, ПЕРЕУСТАНОВИТЬ QUEF
      PTR = QUEF
      QUEF = QUELNK(PTR)
      IF (QUEF .EQ. 0) QUER = 0
      RETURN
      END
```

Рис. 5.12. Операции со связанным стеком.

В заключение этого раздела мы покажем пример совместного использования программ для очереди и стека. Как указывалось выше, стек обычно применяют для хранения списка свободного

```
      SUBROUTINE INSERT (*, DATA, AVLPTR, QUEF, QUER)
      INTEGER LOC, DATA, AVLPTR, QUEF, QUER, LIST, LINK
      COMMON LIST(1000), LINK(1000)
C=================================================================
C ВКЛЮЧЕНИЕ В СВЯЗАННУЮ ОЧЕРЕДЬ  =                                =
C=============================                                   =
C    ПОДПРОГРАММА ВЫПОЛНЯЕТ ВКЛЮЧЕНИЕ В СВЯЗАННУЮ ОЧЕРЕДЬ.       =
C    ОНА СОСТОИТ ИЗ ТРЕХ ЧАСТЕЙ:                                 =
C      1. ПОЛУЧИТЬ УЗЕЛ ИЗ СПИСКА СВОБОДНОЙ ПАМЯТИ;              =
C      2. ЗАПИСАТЬ В НЕГО ДАННЫЕ;                                =
C      3. ПОДСОЕДИНИТЬ УЗЕЛ К ОЧЕРЕДИ.                           =
C    КАК СТЕК СВОБОДНОГО ПРОСТРАНСТВА,ТАК И ОЧЕРЕДЬ СОСТОЯТ      =
C    ИЗ УЗЛОВ, СОДЕРЖАЩИХ ПОЛЯ „LIST"И „LINK".                   =
C    ОБА МАССИВА НАХОДЯТСЯ В COMMON-БЛОКЕ.                       =
C    УКАЗАТЕЛЬ НА СПИСОК СВОБОДНОГО ПРОСТРАНСТВА И УКАЗАТЕЛИ     =
C    НАЧАЛА („QUEF") И КОНЦА („QUER") ОЧЕРЕДИ ПЕРЕДАЮТСЯ         =
C    ИЗ ОСНОВНОЙ ПРОГРАММЫ.                                      =
C    ДАННЫЕ, ЗАПИСЫВАЕМЫЕ В НОВЫЙ УЗЕЛ, ПЕРЕДАЮТСЯ ПЕРЕМЕННОЙ „DATA".  =
C    ЕСЛИ СВОБОДНОГО УЗЛА НЕТ, ТО ВОЗВРАТ RETURN1                =
C                                                                =
C    ПРИ ИСКЛЮЧЕНИИ УЗЛА ИЗ СТЕКА СВОБОДНОГО ПРОСТРАНСТВА        =
C    ОТЛАДКА ОТКЛЮЧЕНА.                                          =
C    ПРОГРАММИСТ: ---                 ДАТА: ---                  =
C=================================================================
C
C. . . . . . . . . ВЗЯТЬ УЗЕЛ ИЗ СПИСКА СВОБОДНОГО ПРОСТРАНСТВА
      CALL LSTPOP (&99, AVLPTR, LOC, .FALSE.)
C. . . . . . . . . ЗАПИСАТЬ ДАННЫЕ В НОВЫЙ УЗЕЛ
      LIST(LOC) = DATA
C. . . . . . . . . ПОДСОЕДИНИТЬ УЗЕЛ К ОЧЕРЕДИ
      CALL QUEADD (&99, QUEF, QUER, LOC)
C. . . . . . . . . КОНЕЦ
      RETURN
C. . . . . . . . . ВЫХОД ПРИ НЕХВАТКЕ В СПИСКЕ СВОБ. ПРОСТР.
   99 RETURN 1
      END
```

Рис. 5.13. Совместное использование очередей и стека.

пространства. На рис. 5.13 дана программа, которая выполняет распределение памяти; она получает узел из списка свободного пространства, помещает в этот узел данные и подсоединяет его к очереди.

5.6. Списки со многими связями

5.6.1. Несколько упорядочений одного списка

Во многих приложениях необходимо просматривать список более чем одним способом. Рассмотрим, например, систему для хранения записей о пациентах больницы. Каждая запись содержит такую информацию, как фамилия пациента, номер палаты, заболевание. В некоторых приложениях нужно, чтобы записи были упорядочены по фамилиям пациентов; в других случаях требуется доступ ко всем пациентам, которых лечит определенный врач. Можно было бы иметь один основной список записей обо всех пациентах в алфавитном порядке и отдельные списки,

в которых для каждого врача хранились бы записи обо всех его пациентах. Такое решение имеет два очевидных недостатка. Во-первых, неэффективно используется память, поскольку каждая запись имеется в двух экземплярах. Во-вторых, трудно поддерживать согласованность всех списков: при всяком изменении записи из основного списка нужно аналогичным образом изменить запись в одном из врачебных списков.

ALFHD

5

DOCHD

1	9
2	5
3	1

	ФАМИЛИЯ	ALFLNK	DOCLNK	Другие поля
1	Хьюз	8	8	
2	Роуз	0	0	
3	О'Кейн	4	4	
4	Пфлигер	2	0	
5	Адамс	9	6	
6	Мэшш	3	3	
7	Маккей	6	2	
8	Кох	7	0	
9	Голдфин	1	7	

Рис. 5.14. Список со многими связями для записей о пациентах.

Для решения этой задачи удобно было бы иметь запись о каждом пациенте в одном экземпляре, но сделать ее частью двух независимых списков: основного и одного из врачебных списков. Возможный подход состоит в том, что строится последовательный список записей, упорядоченный по фамилиям пациентов. Каждая запись, кроме того, должна содержать поле для связи с другими больными, которых лечит тот же врач; для каждого врача нужно иметь указатель начала списка его пациентов. Несмотря на то что такая организация системы решает проблему экономии памяти и согласованности информации в списках, она все же не может нас удовлетворить, поскольку включения и исключения из упорядоченного последовательного списка связаны с перемещением большого объема данных.

Очевидно, что хорошего решения этой задачи нельзя добиться, если имеется только одно поле связи. Имея же два поля связей, можно сделать запись элементом двух независимых связанных списков. На рис. 5.14 показано такое решение. Здесь девять пациентов и три врача. Все пациенты при помощи поля связи ALFLNK и указателя ALFHD связаны в список в алфавитном порядке. (Фамилии упорядочены по английскому алфавиту: Adams, Goldfine, Hughes, Koch, Mackey, Mashey, O'Kane, Pfleeger, Rose.)

Переменная ALFHD указывает на Адамса; за Адамсом идет Голдфин, что указано в поле ALFLNK для Адамса; за Голдфином идет Хьюз и т. д. Имеется также три указателя на врачебные списки, DOCHD (1), DOCHD (2) и DOCHD (3). При помощи поля связи DOCLNK связаны в списки все пациенты, лечащиеся

у одного и того же врача. Так, Голдфина, Маккея и Роуза лечит врач 1; Адамса, Мэши, О'Кейна и Пфлигера лечит врач 2, а Хьюз и Кох лечатся у врача 3.

В предложенном решении в каждой записи имеется два поля связи, но мы вовсе не обязаны ограничиваться именно двумя ссылками; можно добавить их столько, сколько необходимо. В системе для хранения записей о пациентах можно, например, было иметь списки всех пациентов одного пола или списки для каждого типа заболеваний. Нужно учитывать, однако, что с ростом числа связей усложняется работа со списками. Если, скажем, в нашем примере Пфлигер выписывается, то его нужно исключить не только из основного, но и из соответствующего врачебного списка. Пусть запись для Пфлигера была найдена последовательным просмотром основного списка (указатель на его начало находится в ALPHD), тогда предыдущая и следующая записи известны, но только для основного списка. Следующую запись в нужном врачебном списке нетрудно найти при помощи поля DOCLNK. Задача состоит в нахождении предшественника Пфлигера во врачебном списке. В рассматриваемой структуре для этого необходимо просмотреть все врачебные списки.

5.6.2. Списки с двумя связями

Как отмечено выше, наше решение задачи о списке пациентов не удовлетворительно, если записи часто исключаются. Для того чтобы упростить операцию исключения, введем понятие списка с двумя связями — списка, в котором каждый элемент содержит указатели как на следующий, так и на предшествующий элементы. На рис. 5.15 эта идея иллюстрируется применительно к пре-

ALFHD: 5

DOCHD:

1	9
2	5
3	1

	ФАМИЛИЯ	ALFLNK	DOCLNK	DOCPRD	Другие поля
1	Хьюз	8	8	0	
2	Роуз	0	0	7	
3	О'Кейн	4	4	6	
4	Пфлигер	2	0	3	
5	Адамс	9	6	0	
6	Мэши	3	3	5	
7	Маккей	6	2	9	
8	Кох	7	0	1	
9	Голдфин	1	7	0	

Рис. 5.15. Врачебный список с двумя связями.

дыдущему примеру: в запись добавлено поле DOCPRD, указывающее на предыдущую запись во врачебном списке. Новое решение упрощает операцию исключения, но оно требует дополнительного поля в каждой записи.

В рассматриваемой структуре, найдя удаляемую запись в основном списке, можно определить ее точное положение в соответствующем врачебном списке. Следовательно, можно исключить ее, не просматривая записей в других врачебных списках. В общем случае N-я запись исключается при помощи инструкции

DOCLNK (DOCPRD(N))=DOCPRD (N)

Задача усложняется тем, что иногда приходится удалять головной элемент врачебного списка. Примером тому является исключение записи для Хьюза, где DOCPRD(1)=0. Поскольку она первая во врачебном списке, то головной записью врачебного списка 3 должна стать следующая за ней запись. Это достигается при помощи инструкции DOCHD(3)=DOCLNK(1).

5.7. Деревья

Списки со многими связями можно использовать также для представления структур, иерархических по своей природе. В примере с больницей, например, группы больных находятся на попечении своих лечащих врачей. Предположим, что каждая

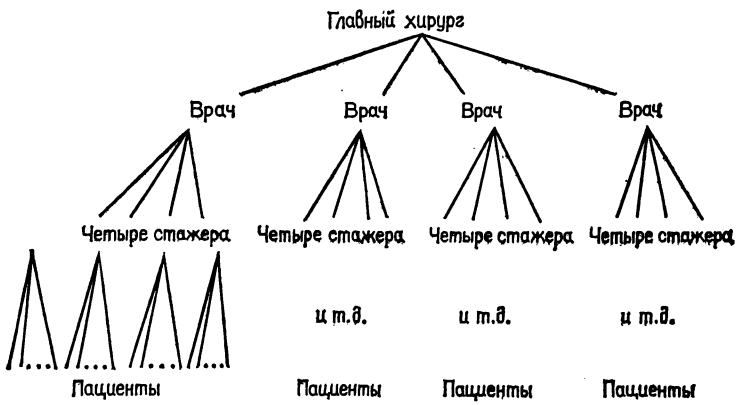


Рис. 5.16. Древовидная структура.

группа из четырех стажеров (всего, скажем, 20 стажеров) подчинена врачу и все врачи подчинены главному хирургу больницы. Такую структуру, называемую деревом, можно изобразить примерно так, как на рис. 5.16.

Структура данных *дерево* состоит из элементов, называемых *узлами*. Узлы соединены друг с другом *направленными дугами*. Если из узла x исходит дуга, ведущая в узел y:

то узел x называется *предшественником* y, а y — *преемником* x. Каждое дерево имеет единственный узел, называемый *корнем*, у которого нет предшественников; любой другой узел имеет ровно одного предшественника. Все узлы, в том числе и корень, имеют ноль или более преемников. Узел, не имеющий преемников, называется *листом*. Каждый узел дерева находится на некотором *уровне*. Корень расположен на уровне 0. Все непосредственные преемники узла, лежащего на уровне j, принадлежат уровню $j+1$.

В дереве на рис. 5.16 корнем является узел, соответствующий главному хирургу. Он является предшественником для четырех узлов — врачей, каждый из которых имеет четырех преемников — узлы, соответствующие стажерам. Узлы-стажеры имеют в качестве своих преемников узлы-пациенты. Листьями этого дерева являются узлы, соответствующие пациентам. Все листья находятся на уровне 3, стажеры — на уровне 2, врачи — на уровне 1, а главный хирург — на уровне 0.

5.7.1. Бинарные деревья

Дерево называется *n-арным деревом*, если каждый его узел имеет не более n преемников. Особый интерес представляют *бинарные деревья*. Бинарным называется 2-арное дерево, преемники которого различаются: один из них является *левым преемником*, а другой — *правым преемником*. Таким образом, узел бинарного дерева может либо вообще не иметь преемников, либо иметь только левого преемника, либо только правого, либо иметь и левого и правого преемников.

Левым (правым) поддеревом узла в бинарном дереве называется максимальное поддерево, имеющее в качестве своего корня левого (правого) преемника этого узла. *Бинарное дерево поиска* — это бинарное дерево, для каждого узла которого все значения данных в левом поддереве меньше, чем значения в узле, а все значения в правом поддереве больше, чем значения в узле. Пример такого дерева изображен на рис. 5.17.

Бинарное дерево из n узлов называется *сбалансированным*, если разность между уровнями любых двух листьев не превышает 1. Дерево на рис. 5.17 несбалансированное (уровень(Z)—уровень(K)=2>1). На рис. 5.18 представлены все возможные сбалансированные деревья поиска, содержащие те же самые значения данных.

Использование бинарных деревьев поиска для реализации таблиц поиска приводит к дополнительному расходу памяти для логических связей (левых и правых) между узлами. Преимущества, даваемые деревьями в этом случае,— простота операций включения и исключения. (Необходимо менять только ссылки.)

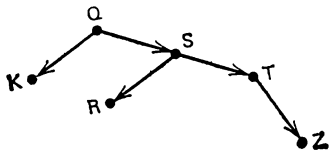


Рис. 5.17. Бинарное дерево поиска.

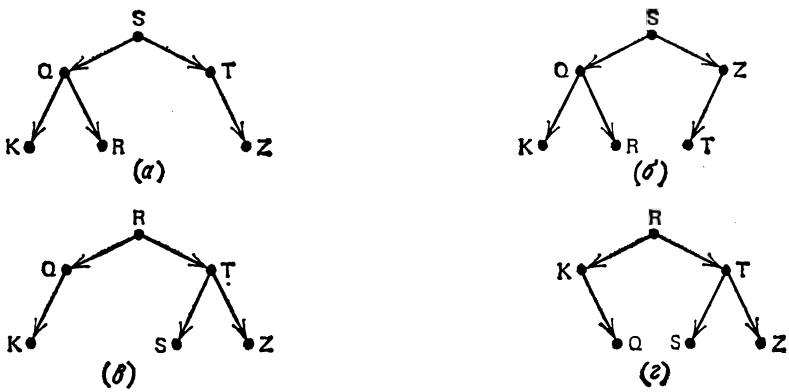


Рис. 5.18. Сбалансированные бинарные деревья.

ROOT

3

	LLINK	DATA	RLINK
1	0	Z	0
2	?	?	?
3	5	Q	8
4	0	T	1
5	0	K	0
6	?	?	?
7	0	R	0
8	7	S	4

Рис. 5.19. Представление бинарного дерева в Фортране.

В Фортране бинарное дерево может быть представлено двумя последовательными списками для левых и правых ссылок и одним или более списком для представления данных в узле. Например, дерево на рис. 5.17 могло бы храниться, как показано на рис. 5.19 (обратите внимание, что позиции 2 и 6 свободны и могут быть использованы для последующих включений в дерево).

Представив бинарное дерево поиска, как на рис. 5.19, мы могли бы задаться вопросом: имеется ли в каком-нибудь узле нашего дерева значение V? Следующий алгоритм служит для решения этой задачи.

1. N ← ROOT.
2. Если N=0, то перейти к 6.
3. Если DATA(N)=V, то перейти к 8.
4. Если DATA(N)>V, то N ← LLINK(N),
   иначе N ← RLINK(N).
5. Перейти к 2.
6. Сообщить, что элемента V нет в таблице.
7. Стоп.
8. Сообщить, что элемент V содержится в позиции N.
9. Стоп.

Этот алгоритм реализован в программе для работы с таблицами символов в следующем разделе.

5.7.2. Пример использования бинарного дерева поиска — работа с таблицей символов

В гл. 0 отмечалось, что имена переменных, встретившиеся в программе на Фортране, запоминаются в таблице символов. Каждый элемент такой таблицы содержит по крайней мере три поля:

имя переменной	адрес	тип

Поле адреса необходимо для того, чтобы поставить в соответствие имени переменной одно или несколько слов памяти. Поле типа нужно, чтобы для каждой инструкции, содержащей эту переменную, сгенерировать правильные машинные команды. (Например, при компиляции инструкции A=B+C требуется проверить типы переменных B и C, чтобы выяснить, применять ли здесь команды целой или вещественной арифметики.)

Для поддержания таблицы символов и работы с ней в компиляторе имеется подпрограмма поиска по таблице. При входе в подпрограмму ей в качестве параметра передается символ, который нужно найти. Если символ есть в таблице, то подпрограмма возвращает указатель на узел, содержащий этот символ. Если

найти символ не удалось, то в таблице символов создается новый элемент для этой переменной. Подпрограмма возвращает указатель на вновь созданный узел и признак того, что он новый.

Поскольку обращения к таблице символов происходят часто, то ее структура и алгоритмы работы с ней должны быть тщательно продуманы. Часто используется описанный в гл. 2 метод хеширования. Другой распространенный подход состоит в том, чтобы хранить элементы таблицы как узел бинарного дерева поиска. На рис. 5.20 представлена подпрограмма TRETAB, иллюстрирующая этот способ.

Дерево, создаваемое при помощи программы TRETAB, не обязательно сбалансированное. Если символы включаются в таблицу в алфавитном порядке, то получится дерево, как на рис. 5.21, *а*; поиск по такому дереву происходит не быстрее, чем

```
      SUBROUTINE TRETAB (VARNAM, NEW, P)
C= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C= ПОДПРОГРАММА ДЛЯ ПОИСКА ПО БИНАРНОМУ ДЕРЕВУ =                    =
C= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C=                                                                  =
C= ПОДПРОГРАММА ДЛЯ ПОИСКА ИМЕНИ VARNAM В ДЕРЕВЕ, НА КОРЕНЬ        =
C= КОТОРОГО УКАЗЫВАЕТ ROOT: ЕСЛИ ИМЯ НЕ НАЙДЕНО, ТО СОЗДАЕТСЯ      =
C= НОВЫЙ ЭЛЕМЕНТ. В ДЕРЕВЕ МОЖЕТ БЫТЬ ДО 100 ЭЛЕМЕНТОВ.            =
C= ПАМЯТЬ РАСПРЕДЕЛЯЕТСЯ ПОСЛЕДОВАТЕЛЬНО, Т. Е. ИСКЛЮЧАТЬ           =
C= ИЗ ТАБЛИЦЫ НЕЛЬЗЯ (FREE В НАЧАЛЕ ДОЛЖНО БЫТЬ 1). ПРИ ПЕРЕ-       =
C= ПОЛНЕНИИ ТАБЛИЦЫ ПЕЧАТАЕТСЯ СООБЩЕНИЕ И ВЫПОЛНЕНИЕ               =
C= ПРОГРАММЫ ПРЕКРАЩАЕТСЯ.                                          =
C=                                                                  =
C=              *** ПЕРЕМЕННЫЕ В COMMON-БЛОКЕ ***                   =
C= ROOT     — УКАЗАТЕЛЬ НА КОРНЕВОЙ УЗЕЛ ДЕРЕВА;                    =
C= LEFT     — СПИСОК УКАЗАТЕЛЕЙ НА ЛЕВЫХ ПРЕЕМНИКОВ;               =
C= RIGHT    — СПИСОК УКАЗАТЕЛЕЙ НА ПРАВЫХ ПРЕЕМНИКОВ;              =
C= NAME     — ИМЯ ПЕРЕМЕННОЙ, ХРАНИМОЙ В ДАННОМ УЗЛЕ;              =
C= MODE     — ТИП ПЕРЕМЕННОЙ;                                       =
C= LOC      — АДРЕС ПЕРЕМЕННОЙ;                                     =
C= FREE     — УКАЗАТЕЛЬ НА ПЕРВЫЙ СВОБОДНЫЙ ЭЛЕМЕНТ;                =
C= NULL     — ЗНАЧЕНИЕ ПУСТОГО УКАЗАТЕЛЯ;                           =
C= OUT      — НОМЕР КАНАЛА ДЛЯ ПЕЧАТИ.                              =
C=                                                                  =
C=              *** ФОРМАЛЬНЫЕ ПАРАМЕТРЫ ***                        =
C= VARNAM — ИМЯ, КОТОРОЕ НУЖНО НАЙТИ;                               =
C= NEW    — УСТАНАВЛИВАЕТСЯ РАВНЫМ .TRUE. , ЕСЛИ ИМЯ РАНЕЕ         =
C=          НЕ ЗАНОСИЛОСЬ В ТАБЛИЦУ, И .FALSE. В ПРОТИВНОМ          =
C=          СЛУЧАЕ;                                                 =
C= P      — УКАЗАТЕЛЬ НА ЭЛЕМЕНТ, В КОТОРОМ НАХОДИТСЯ              =
C=          VARNAM.                                                 =
C=          ПРОГРАММИСТ: — — —          ДАТА: — — —                 =
C= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

      INTEGER LEFT(100), LOC(100), MODE(100), RIGHT(100)
      INTEGER P, ROOT, FREE, PTR, OUT, NULL
      CHARACTER*6 NAME(100), VARNAM
      LOGICAL NEW
      COMMON ROOT, LEFT, RIGHT, NAME, MODE, LOC, FREE, NULL, OUT

      ... ТАБЛИЦА ПУСТА?...
      IF (ROOT .EQ. NULL) GO TO 80
      PTR = ROOT
```

Рис. 5.20. Таблица символов, имеющая структуру дерева. (Текст в инстр. 1000: (5X, ' ** ПЕРЕПОЛНЕНИЕ В TRETAB **'))

```
C     ... СРАВНИТЬ ЭЛЕМЕНТ С VARNAM...
10    IF (NAME(PTR) .EQ. VARNAM) GO TO 40
      IF (NAME(PTR) .LT. VARNAM) GO TO 50
C
C     ... ЛЕВЫЙ ПРЕЕМНИК...
      IF (LEFT(PTR) .EQ. NULL) GO TO 30
      PTR = LEFT(PTR)
      GO TO 10
C     ... ДОБАВИТЬ УЗЕЛ СЛЕВА...
30    LEFT(PTR) = FREE
      GO TO 70
C
C     ... ЭЛЕМЕНТ НАЙДЕН...
40    P = PTR
      NEW = .FALSE.
      RETURN
C
C     ... ПРАВЫЙ ПРЕЕМНИК...
50    IF (RIGHT(PTR) .EQ. NULL) GO TO 60
      PTR = RIGHT(PTR)
      GO TO 10

C     ... ДОБАВИТЬ УЗЕЛ СПРАВА...
60    RIGHT(PTR) = FREE
C
C     ... ПРОВЕРКА НА ПЕРЕПОЛНЕНИЕ ТАБЛИЦЫ...
70    IF (FREE .LE. 100) GO TO 99
      WRITE (OUT,1000)
1000  FORMAT (5X, '** OVERFLOW IN TRETAB **')
      STOP
C
C     ... ПОМЕСТИТЬ НОВЫЙ УЗЕЛ В ПОЗИЦИЮ P...
80    ROOT = FREE
99    P = FREE
      NAME(P) = VARNAM
      LEFT(P) = NULL
      RIGHT(P) = NULL
      FREE = FREE + 1
      NEW = .TRUE.
      RETURN
      END
```

Продолжение Рис. 5.20.

последовательный поиск в упорядоченной таблице. Если же символы включаются в порядке D, B, F, A, C, E, G, то получается дерево, изображенное на рис. 5.21, *б*; поиск по такому дереву осуществляется с той же скоростью, что и бинарный поиск в упорядоченной таблице. Поведение бинарного дерева поиска зависит от того, сбалансированное оно или нет. Для поиска по сбалансированному дереву из n узлов достаточно просмотреть не более $\log_2(n)+1$ узлов. Если же дерево сильно несбалансировано, то при поиске может потребоваться проверка всех n узлов. Существуют алгоритмы, сохраняющие сбалансированность бинарного дерева поиска при включениях и исключениях, но их рассмотрение не входит в круг вопросов данной книги.

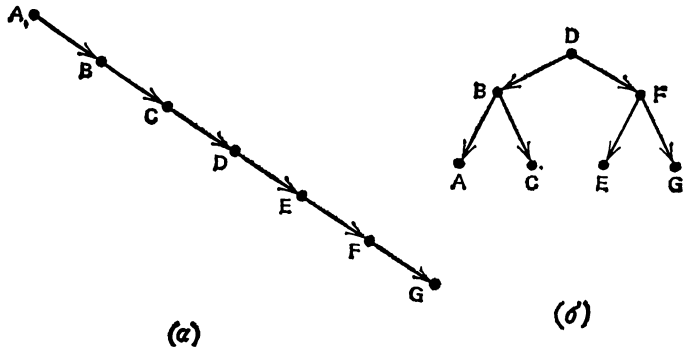


Рис. 5.21. Деревья поиска; несбалансированное и сбалансированное.

5.8. Пример работы со списками — моделирование автозаправочной станции

5.8.1. О моделировании

Моделирование — это имитация всех или некоторых аспектов поведения одной системы при помощи другой, не похожей системы. Моделирование осуществляется часто с целью глубже понять систему или выяснить, как возможные изменения могут повлиять на ее поведение. К моделированию прибегают в тех случаях, когда прямой эксперимент либо невозможен (новая, еще не существующая система), либо неэкономичен (слишком дорого обойдется создание реальной системы), либо «аморален» (крушение самолета с пилотом при посадке), либо слишком продолжителен (эксперименты с лесным массивом и другими экологическими системами).

В большинстве случаев при моделировании можно предположить, что время изменяется дискретными шагами от одного события до другого. Взяв за основу события в системе, мы часто получаем возможность промоделировать продолжительные реальные отрезки времени за очень малое время. Например, основные события, происходящие за 10-часовой рабочий день на автозаправочной станции, можно смоделировать за несколько секунд.

5.8.2. Моделируемая система

Мы проиллюстрируем использование моделирования на примере имитации работы автозаправочной станции. Цель моделирования — дать возможность владельцу предсказывать прибыль и, возможно, изменять организацию работы на станции с целью увеличения прибыли.

На моделируемой станции имеется один бокс для технического обслуживания автомобилей и две колонки. К каждому из этих трех объектов имеется отдельный подъезд для автомобилей, и на каждом из них работает один человек, который производит соответствующее обслуживание клиентов. Обе колонки работают с 8 часов утра до 6 вечера, механик в боксе работает с 9 утра до 6 вечера.

Каждый час начиная с 9 часов утра на станцию прибывает один автомобиль, требующий технического обслуживания. Автомобили, требующие заправки, прибывают каждые три минуты в часы пик (с 8 до 9 утра и с 4 до 6 вечера) и каждые 6 минут в остальные часы (с 9 утра до 4 дня). Водитель автомобиля, прибывшего для заправки, выбирает ту из двух очередей к бензоколонкам, в которой в данный момент меньше автомобилей. После каждого прибытия автомобиля на заправку можно вычислить время следующего прибытия. Интервал от одного прибытия до другого вычисляется по формуле

$$интервал = -ти * \log_2(r),$$

где r принимает случайные значения, равномерно распределенные между 0 и 1, а *ти* — среднее время между прибытиями (3 или 6 минут). Такой метод генерирования интервалов называется методом с отрицательным показателем распределения интервалов. Он обладает тем свойством, что малые значения интервалов встречаются чаще, чем большие. (Примерно 63% всех прибытий происходит с интервалом меньше *ти*, менее 5% интервалов превышают значение 3**ти*.)

Перечислим еще некоторые свойства моделируемой системы:

1. На станции может одновременно находиться не более 25 автомобилей. Так, если общее число автомобилей в очередях плюс число обслуживаемых автомобилей превышает 25, то водители вновь прибывающих автомобилей предпочитают проехать до следующей станции.

2. Для заправки автомобиля требуется от 0 до 20 галлонов [1]) бензина порциями по 5 галлонов (т. е. автомобиль может получить 0, 5, 10, 15 или 20 галлонов). Цена бензина — 75 центов за галлон. Автомобиль может получить также 0 или 1 кварту [2]) масла по цене 85 центов за кварту. Еще два вида услуг — мытье ветрового стекла и подкачка шин — производятся бесплатно.

3. В ремонтном боксе на автомобиль можно установить от 0 до 4 шин по цене 34 доллара за штуку; замена масла стоит 7 дол-

---

[1]) 1 галлон=3,78 л.
[2]) 1 кварта=1,14 л.

ларов 50 центов, смазка — 3 доллара 50 центов; общая регулировка автомобиля стоит 37 долларов 95 центов.

4. При обслуживании автомобиля на бензоколонке затрачивается по 2 минуты на заливание каждых 5 галлонов бензина, 3 минуты на добавление кварты масла, 2 минуты на мытье ветрового стекла и 1 минута на подкачку шин.

5. При обслуживании в ремонтном боксе затрачивается 9 минут на установку одной шины, 20 минут на замену шины, 10 минут на смазку, 30 минут на регулировку автомобиля.

6. Прибыль исчисляется в размере 20% общей выручки.

5.8.3. Планирование и обработка событий

При моделировании работы станции возможны шесть типов событий, которые необходимо планировать и обрабатывать:

1. Отбытие автомобиля с линии обслуживания 1 (первая бензоколонка).
2. Отбытие автомобиля с линии обслуживания 2 (вторая бензоколонка).
3. Отбытие автомобиля с линии обслуживания 3 (ремонтный бокс).
4. Прибытие автомобиля на заправку.
5. Прибытие автомобиля, требующего технического обслуживания.
6. Запрос о текущем состоянии станции обслуживания.

События типа 5 происходят ровно через час начиная с 9 утра. После каждого прибытия планируется следующее прибытие через 1 час.

События типа 4 планируются со средним интервалом 3 или 6 минут в зависимости от времени дня. Прибытие первого автомобиля на заправку планируется через один интервал для часов пик от начала работы станции (т. е. вскоре после 8 утра).

События типа 1, 2 и 3 планируются в начале обслуживания автомобиля. Время отбытия зависит от общего времени, необходимого для выполнения всех требуемых услуг.

События типа 6 не соответствуют никаким событиям в реальной системе. Они помогут нам проанализировать, насколько эффективно работает станция. Первый отчет выдается в 8 утра, а последующие отчеты печатаются через каждый час до 6 часов вечера.

Если два события должны произойти в один и тот же момент, то первым обрабатывается событие, номер типа которого меньше.

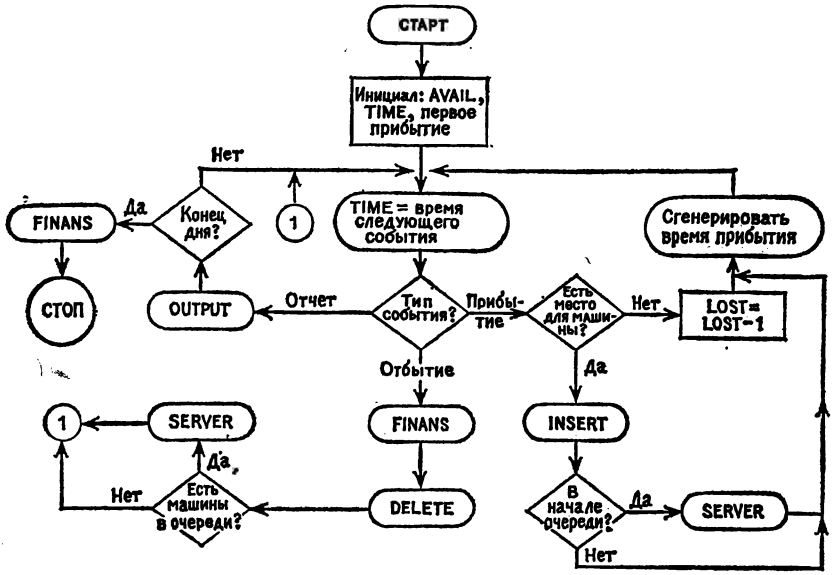


Рис. 5.22. Блок-схема моделирования автозаправочной станции.

Так, отбытие автомобиля от колонки 1 обрабатывается в первую очередь, а печать отчета — в последнюю.

Блок-схема программы для моделирования событий на станции обслуживания изображена на рис. 5.22; сама программа приведена на рис. 5.23. В программе используется последовательный список TF для представления моментов наступления событий каждого типа. Массив TF состоит из шести элементов, где TF(I) — время наступления следующего события типа I. В любой момент времени TIME для обработки выбирается событие типа I, если TF(I) содержит минимальное время, большее или равное TIME. При $I \leqslant 3$ после окончания обработки отбытия мы устанавливаем TF(I)=TF(I)—1. Таким образом, событие I блокируется до тех пор, пока не будет вычислено время следующего отбытия с этой линии. При I=4 или 5 после обработки текущего прибытия элементу TF(I) присваивается время следующего прибытия. При I=6 печатается отчет о состоянии системы и планируется выдача следующего отчета ровно через час.

Выдача на печать от моделирующей программы изображена на рис. 5.24. Изображена лишь часть отчета. А именно приведена информация о деятельности системы до 9 утра, отчет о состоянии станции на 11 часов утра и полный отчет о работе после 5 час. 45 мин. вечера.

```
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C     . . .МОДЕЛИРОВАНИЕ АВТОЗАПРАВОЧНОЙ СТАНЦИИ. . .                =
C                                                                      =
C ЦЕЛЬ: В ПРОГРАММЕ МОДЕЛИРУЕТСЯ РАБОЧИЙ ДЕНЬ ЗАПРАВОЧНОЙ=
C     СТАНЦИИ. ПОКАЗАНО ИСПОЛЬЗОВАНИЕ МАССИВОВ ДЛЯ ПРЕДСТАВ-=
C     ЛЕНИЯ СТЕКОВ И ОЧЕРЕДЕЙ. ДЛЯ ГЕНЕРАЦИИ НЕПРЕДСКАЗУЕ-=
C     МЫХ МОМЕНТОВ ПРИБЫТИЯ И ЗАКАЗОВ НА ОБСЛУЖИВАНИЕ ПРИ-=
C     МЕНЕН ГЕНЕРАТОР СЛУЧАЙНЫХ ЧИСЕЛ.                                =
C ОСНОВНЫЕ ПЕРЕМЕННЫЕ:                                                =
C     LANE    — ГОЛОВЫ СВЯЗАННЫХ СПИСКОВ МАШИН, ЖДУЩИХ ОБ-=
C                 СЛУЖИВАНИЯ;                                        =
C     TF      — МОМЕНТЫ ОТБЫТИЯ (ЛИНИИ 1—3), ПРИБЫТИЯ (БЕНЗИН,=
C                 РЕМОНТ) И ОТЧЕТА;                                  =
C     CELLS   — МАССИВ ОПИСАТЕЛЕЙ ДЛЯ 25 МАШИН;                      =
C     LINK    — МАССИВ ССЫЛОК В 'CELLS'. ЭКВИВАЛЕНТЕН СТОЛБЦУ=
C                 6 'CELLS':                                         =
C     AVAIL   — ГОЛОВА СПИСКА СВОБОДНЫХ ПОЗИЦИЙ В 'CELLS';           =
C     TIME    — ТЕКУЩЕЕ ВРЕМЯ В МОДЕЛИ СИСТЕМЫ;                       =
C     LOST    — (ОТРИЦАТЕЛЬНОЕ) ЧИСЛО МАШИН, УЕХАВШИХ БЕЗ=
C                 ОБСЛУЖИВАНИЯ.                                      =
C ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                            =
C     ALLOC   — ВЗЯТЬ СВОБОДНЫЙ ЭЛЕМЕНТ МАССИВА 'CELLS';              =
C     NEXT    — ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СОБЫТИЕ;                         =
C     DELETE  — ИСКЛЮЧИТЬ ЭЛЕМЕНТ ИЗ НАЧАЛА НЕКОТОРОЙ ОЧЕРЕДИ;=
C     INSERT  — ВКЛЮЧИТЬ ЭЛЕМЕНТ В КОНЕЦ НЕКОТОРОЙ ОЧЕРЕДИ;          =
C     RAND    — ФУНКЦИЯ ДЛЯ ПОЛУЧЕНИЯ СЛУЧАЙНОГО ЧИСЛА;              =
C     SERVER  — ВЫЧИСЛИТЬ ВРЕМЯ, НУЖНОЕ ДЛЯ ОБСЛУЖИВАНИЯ;            =
C     DEMANS  — ОПРЕДЕЛИТЬ УСЛУГИ, НЕОБХОДИМЫЕ ДЛЯ ОЧЕРЕДНОГО=
C                 КЛИЕНТА;                                           =
C     FINANS  — ВЫЧИСЛИТЬ СТОИМОСТЬ ЗАКАЗАННЫХ УСЛУГ И СУММУ;=
C     ARRIVE  — ВЫЧИСЛЕНИЕ МОМЕНТА СЛЕДУЮЩЕГО ПРИБЫТИЯ;              =
C     SHORT   — ВЫДАЕТ НОМЕР БОЛЕЕ КОРОТКОЙ ОЧЕРЕДИ НА=
C                 ЗАПРАВКУ;                                          =
C     OUTPUT  — ПЕЧАТАЕТ ОТЧЕТ О ТЕКУЩЕМ СОСТОЯНИИ                   =
C     ПРОГРАММИСТ: — — —                ДАТА: — — —                  =
C = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

      INTEGER LANE(3), TF(6), TIME, CUTOFF, PRT
      INTEGER CELLS(25,6),AVAIL,LINK(25),LOST,ALLOC,ARRIVE,SHORT
      REAL BILL
      EQUIVALENCE (CELLS(1,6),LINK(1))
      DATA LOST/0/,TIME/800/,LANE/3*0/,TF/5*759,800/,CUTOFF/1800/
      DATA AVAIL/1/,PRT/6/
C
C     * * * ИНИЦИАЛИЗАЦИЯ СПИСКА СВОБОДНОГО ПРОСТРАНСТВА
      DO 1 I=1,24
         LINK(I) = I+1
    1 CONTINUE
      LINK(25) = 0
      WRITE (PRT,100)
  100 FORMAT ('1',15X,'** SERVICE STATION SIMULATION **' / )
C
```

Рис. 5.23. Программа моделирования.

(В главной программе **100**: ('1', 15X, '** МОДЕЛИРОВАНИЕ СТАНЦИИ ОБСЛУЖИВАНИЯ **'));

101: (5X, ' ОТБЫТИЕ ИЗ ОЧЕРЕДИ', I2, 'МАШИНА НОМЕР', I3, ' СЧЕТ $ ', F 6. 2, 'ВРЕМЯ', I5);

102: (5X, ' ПРИБЫТИЕ В ОЧЕРЕДЬ', I2, 'МАШИНА НОМЕР', I3, 15X, 'ВРЕМЯ', I5).

В процедуре ALLOC **100**: (5X, ' ** СТАНЦИЯ ПЕРЕПОЛНЕНА, КЛИЕНТ УЕХАЛ НЕОБСЛУЖЕННЫМ').

В подпрограмме NEXT **100**: (5X, ' **ОШИБКА: НЕТ СОБЫТИЯ — ОСТАНОВ В ПОДПРОГРАММЕ NEXT **').

В подпрограмме DELETE **100**: (5X, ' **ОШИБКА: ИСКЛЮЧЕНИЕ ИЗ ПУСТОЙ ОЧЕРЕДИ **').

В подпрограмме FINANS **101**: ('0', T23, ' ** ИТОГ ДНЯ** '); **102**: ('ОБСЛУЖЕНО', I4, 'КЛИЕНТОВ, СРЕДНИЙ СЧЕТ СОСТАВИЛ $', F62, ' СУММА ПРИБЫЛИ= $', F 8.2/,I4, 'КЛИЕНТА НЕ ОБСЛУЖЕНЫ').

В подпрограмме OUTPUT **100**: ('0', I4X, 'СОСТОЯНИЕ ЗАПРАВОЧНОЙ СТАНЦИИ: ', I5, 'ЧАСОВ');

101: PUMP — КОЛОНКА, GARAGE — РЕМОНТ.

```
C     . . . ЗАПЛАНИРОВАТЬ ПЕРВЫЕ ПРИБЫТИЯ: ЗАПРАВКА И РЕМОНТ. . .
      TF(4) = ARRIVE(TIME,4)
      I = 5
C
C     . . . ВРЕМЯ НОВОГО ПРИБЫТИЯ: ЗАПРАВКА ИЛИ РЕМОНТ. . .
   10 TF(I) = ARRIVE(TIME,I)
C
C     . . . КАКОЕ СОБЫТИЕ ПРОИЗОЙДЕТ СЛЕДУЮЩИМ? . . .
   20 CALL NEXT(TIME,TF,I)
      IF(I .NE. 6) GO TO 25
C
C     . . . ОТЧЕТ, ЕСЛИ КОНЕЦ ДНЯ — СТОП, ИНАЧЕ — ЗАПЛАН. НОВЫЙ ОТЧЕТ. . .
        CALL OUTPUT(TIME,LANE,CELLS,LINK)
        IF(TIME .GE. CUTOFF) GO TO 60
        TF(6) = TF(6)+101
        GO TO 20
   25 IF(I .GT. 3) GO TO 30
C
C     . . . ОТБЫТИЕ: ОБСЛУЖИТЬ СЛЕДУЮЩЕГО КЛИЕНТА ИЗ ОЧЕРЕДИ. . .
      CALL FINANS(LANE(I),I,CELLS,BILL)
      WRITE (PRT,101) I,CELLS(LANE(I),5),BILL,TIME
  101 FORMAT (5X,'EXIT  FROM LANE', I2,' CAR #',I3,' BILL WAS $',F6.2,
     1   '  AT TIME',I5)
      CALL DELETE(LANE(I),LINK,AVAIL)
      IF(LANE(I) .NE. 0) CALL SERVER(I,TIME,CELLS,LANE,TF)
      GO TO 20

C
C     . . . НОВОЕ ПРИБЫТИЕ. . . ЕСТЬ МЕСТО?
   30 K = ALLOC(AVAIL,LINK)
      IF(K .NE. 0) GO TO 40
C     . . . НЕТ МЕСТА. . .
        LOST=LOST-1
        GO TO 10
   40 CALL DEMANS(K,CELLS)
C     . . . ЗАПРАВКА ИЛИ РЕМОНТ?
      J = 3
      IF(I .EQ. 5) GO TO 50
C     . . . ЗАПРАВКА: ВСТАТЬ В КРАТЧАЙШУЮ ОЧЕРЕДЬ. . .
      J = SHORT(LINK,LANE)
C     . . . ВСТАВИТЬ ВНОВЬ ПРИБЫВШЕГО В ОЧЕРЕДЬ. . .
   50 CALL INSERT(K,LANE(J),LINK)
      WRITE(PRT,102) J,CELLS(K,5),TIME
  102 FORMAT(5X,'ENTER INTO LANE',I2,' CAR #',I3,19X,'AT TIME',I5)
      IF(LINK(LANE(J)) .EQ. 0) CALL SERVER(J,TIME,CELLS,LANE,TF)
      GO TO 10
C
C     . . . КОНЕЦ ДНЯ. . .
   60 CALL FINANS(LOST,I,CELLS,BILL)
      STOP
      END

      INTEGER FUNCTION ALLOC(AVAIL,LINK)
C=====================================================================
C ВЫДЕЛИТЬ СВОБОДНУЮ СТРОКУ В МАССИВЕ CELLS.                          =
C                                                                     =
C AVAIL — УКАЗАТЕЛЬ НА ПЕРВУЮ СВОБОДНУЮ ПОЗИЦИЮ.                      =
C LINK  — ПОЛЕ ССЫЛКИ В СПИСКЕ СВОБОДНОГО ПРОСТРАНСТВА.               =
C                                                                     =
C ВОЗВРАЩАЕТСЯ УКАЗ. НА ПЕРВУЮ СВОБ. ПОЗИЦИЮ ИЛИ 0 ЕСЛИ НЕТ МЕСТА.    =
C=====================================================================
      INTEGER AVAIL,LINK(25),PRT
      DATA PRT/6/
C
C     . . . ВЫДЕЛИТЬ СВОБОДНОЕ МЕСТО ИЛИ ОТКАЗАТЬ. . .
      IF(AVAIL .NE. 0) GO TO 10
      WRITE(PRT,100)
  100 FORMAT(5X,'** FULL STATION: CUSTOMER DROVE AWAY WITHOUT SERVICE')
      ALLOC = 0
      RETURN
C
```

Продолжение Рис. 5.23.

```
C       . . . ПРОДВИНУТЬ УКАЗАТЕЛЬ 'AVAIL'
   10 ALLOC = AVAIL
      AVAIL = LINK(AVAIL)
      RETURN
      END

      SUBROUTINE NEXT(TIME,TF,EVTYP)
C=====================================================================
C ОПРЕДЕЛИТЬ ВРЕМЯ И ТИП СЛЕДУЮЩЕГО СОБЫТИЯ.                          =
C                                                                    =
C TIME  — НА ВХОДЕ-ТЕКУЩЕЕ ВРЕМЯ; НА ВЫХОДЕ-ВРЕМЯ ОЧЕРЕДНОГО СОБЫТИЯ.  =
C TF    —МОМЕНТЫ ОЧЕРЕДНЫХ СОБЫТИЙ КАЖДОГО ТИПА.                     =
C EVTYP —ТИП СЛЕДУЮЩЕГО СОБЫТИЯ.                                     =
C=====================================================================
      INTEGER TIME,TF(6),EVTYP,EVTIME,PRT
      DATA PRT/6/
C
C       . . . НАЙТИ „МИНИМАЛЬНОЕ“ СОБЫТИЕ „НЕ МЕНЬШЕ“ TIME
      EVTYP = 0
      EVTIME = 2401
      DO 10 J = 1,6
         IF((TF(J) . GE. EVTIME) .OR. (TF(J) .LT. TIME)) GO TO 10
         EVTYP = J
         EVTIME = TF(J)
   10 CONTINUE
       IF (EVTYP .NE. 0) GO TO 20
C       . . . ЕСЛИ НЕТ СЛЕДУЮЩЕГО СОБЫТИЯ, ВОЗНИКАЮТ ТРУДНОСТИ
      WRITE(PRT,100)
  100 FORMAT(5X,'**ERROR: NO NEXT EVENT - STOPPING IN NEXT**')
      STOP
C
C       . . . УСТАНОВ. TIME И УМЕНЬШ. TF(EVTYP), ЧТОБЫ НЕ ВЫБР. ЕГО СНОВА
   20 TIME = TF(EVTYP)
      TF(EVTYP) = TF(EVTYP)-1
      RETURN
      END


      SUBROUTINE DELETE(LIST,LINK,AVAIL)
C=====================================================================
C ИСКЛЮЧИТЬ ПЕРВЫЙ ЭЛЕМЕНТ СПИСКА И ВЕРНУТЬ В СВОБ. ПРОСТРАНСТВО      =
C                                                                    =
C LIST — ГОЛОВА СПИСКА, ИЗ КОТОРОГО ИСКЛЮЧАЕТСЯ ЭЛЕМЕНТ              =
C LINK — ПОЛЕ СВЯЗИ В ЭЛЕМЕНТАХ СПИСКА И СВОБОДНОГО ПРОСТРАНСТВА      =
C AVAIL—ГОЛОВА СПИСКА СВОБОДНОГО ПРОСТРАНСТВА                        =
C=====================================================================
      INTEGER AVAIL,LINK(25),LIST,PTR,PRT
      DATA PRT/6/
C
C       . . . СПИСОК ПУСТОЙ? . . .
      IF(LIST .GT. 0) GO TO 10
      WRITE(PRT,100)
  100 FORMAT(5X,'**ERROR: TRIED TO DELETE FROM EMPTY QUEUE**')
      STOP
C
C       . . . УДАЛИТЬ ПЕРВЫЙ ЭЛЕМЕНТ 'LANE(I)' И ВЕРНУТЬ В 'AVAIL'
   10 PTR = LINK(LIST)
      LINK(LIST) = AVAIL
      AVAIL = LIST
      LIST = PTR
      RETURN
      END

      SUBROUTINE INSERT(K,LIST,LINK)
C=====================================================================
C ВКЛЮЧИТЬ ЭЛЕМЕНТ В КОНЕЦ ОЧЕРЕДИ                                    =
C                                                                    =
C K      - УКАЗАТЕЛЬ НА ЭЛЕМЕНТ, КОТОРЫЙ НУЖНО ВКЛЮЧИТЬ              =
C LIST   - ГОЛОВА СПИСКА, В КОТОРЫЙ ВКЛЮЧАЕТСЯ ЭЛЕМЕНТ              =
C LINK   - ПОЛЕ ССЫЛКИ В ЭЛЕМЕНТАХ СПИСКА                            =
C=====================================================================
      INTEGER LINK(25),LIST,K,PTR,PRT
      DATA PRT/6/
C
```

Продолжение Рис. 5.23.

```
C     . . . НАЙТИ КОНЕЦ ОЧЕРЕДИ. . .
      IF(LIST .EQ. 0) GO TO 30
      PTR = LIST
   10 IF(LINK(PTR) .EQ. 0) GO TO 20
      PTR = LINK(PTR)
      GO TO 10
C     . . . ДОБАВИТЬ ЭЛЕМЕНТ 'K' В КОНЕЦ ОЧЕРЕДИ. . .
   20 LINK(PTR) = K
      GO TO 40
C
C     . . . СОЗДАТЬ НОВЫЙ СПИСОК ИЗ ОДНОГО ЭЛЕМЕНТА. . .
   30 LIST = K
   40 LINK(K) = 0
      RETURN
      END

      REAL FUNCTION RAND(RMAX)
C=====================================================================
C ВЫЧИСЛЕНИЕ СЛУЧ. ВЕЩ. ЧИСЛА В ДИАПАЗОНЕ ОТ 0 ДО „RMAX" (НЕ ВКЛЮЧИТЕЛЬНО).=
C ПРОГРАММА ПРИГОДНА ДЛЯ МАШИН С 32-РАЗРЯДНЫМ ДОПОЛНИТЕЛЬНЫМ КОДОМ,      =
C В ЧАСТНОСТИ, IBM 360-370.                                            =
C=====================================================================
C
      REAL RMAX
      INTEGER IRAND, MULT, LARGE
      DATA IRAND/137462873/, MULT/65539/, LARGE/2147483647/
C     . . . ВЫЧИСЛИТЬ НОВОЕ СЛУЧ. ЦЕЛОЕ (ОТ 1 ДО 2 ** 31-1)
      IRAND = IRAND*MULT
      IF(IRAND .LT. 0) IRAND = (IRAND+LARGE)+1
C     . . . ВЫЧИСЛИТЬ ТРЕБУЕМОЕ ВЕЩЕСТВЕННОЕ ЧИСЛО
      RAND = RMAX*IRAND*.4656613E-9
      RETURN
      END
```

```
      SUBROUTINE SERVER(I,TIME,CELLS,LANE,TF)
C=====================================================================
C ВЫЧИСЛИТЬ ОЖИДАЕМОЕ ВРЕМЯ ОБСЛУЖ. КЛИЕНТА ИЗ ОЧЕРЕДИ LANE(I),      =
C ЧТОБЫ ОПРЕДЕЛИТЬ ВРЕМЯ ОТБЫТИЯ МАШИНЫ.                              =
C ВРЕМЯ ОБСЛУЖИВАНИЯ:                                                 =
C ЗАПРАВКА: 5 ГАЛЛОНОВ-2 МИН., МАСЛО-3 МИН., МЫТЬЕ-2 МИН., ПОДКАЧКА-1 МИН.=
C РЕМОНТ: ШИНА-9 МИН., МАСЛО-20 МИН., СМАЗКА-10 МИН., РЕГУЛИРОВКА-30 МИН.=
C                                                                     =
C I      - НОМЕР ОЧЕРЕДИ-1 ИЛИ 2 ДЛЯ ЗАПРАВКИ, 3 ДЛЯ РЕМОНТА;         =
C TIME   - ТЕКУЩЕЕ ВРЕМЯ;                                             =
C CELLS- ЭЛЕМЕНТЫ, ОПИСЫВАЮЩИЕ ВИД И ОБЪЕМ УСЛУГ;                     =
C LANE   - ГОЛОВЫ СПИСКОВ-ОЧЕРЕДЕЙ 1,2,3;                             =
C TF     - ВРЕМЕНА СЛЕДУЮЩИХ СОБЫТИЙ КАЖДОГО ТИПА.                    =
C=====================================================================
      INTEGER TIME,CELLS(25,6),LANE(3),SRVTIM,TF(6),HR,TIMES(4,2),I
      DATA TIMES/2,3,2,1,9,20,10,30/
C
C     . . . ЗАПРАВКА ИЛИ РЕМОНТ?
      J = 1
      IF(I .EQ. 3) J = 2
C
C     . . . ВЫЧИСЛИТЬ ОБЩЕЕ ВРЕМЯ ОБСЛУЖИВАНИЯ. . .
      SRVTIM = 0
      DO 10 K=1,4
         SRVTIM = SRVTIM+TIMES(K,J)*CELLS(LANE(I),K)
   10 CONTINUE
C
C     . . . ВЫЧИСЛИТЬ ВРЕМЯ ОКОНЧАНИЯ ОБСЛУЖИВАНИЯ. . .
      HR = TIME/100
      MIN = TIME-HR*100+SRVTIM
      HR = HR+MIN/60
      MIN = MIN-(MIN/60)*60
      TF(I) = HR*100+MIN
      RETURN
      END
```

Продолжение Рис. 5.23.

```
      SUBROUTINE DEMANS(K,CELLS)
C=====================================================================
C ВЫЧИСЛЯЕТ УСЛУГИ ДЛЯ ОЧЕРЕДНОЙ МАШИНЫ                              =
C                                                                    =
C ИСПОЛЬЗУЕТ ЦЕЛОЧИСЛ. СЛУЧАЙНУЮ ВЕЛИЧИНУ                            =
C С РАВНОМ. РАСПРЕД. И ЗНАЧЕНИЯМИ:                                   =
C ЗАПРАВКА: 0-4 ПОРЦ. ПО 5 ГАЛ.; МАСЛО, МЫТЬЕ, ПОДКАЧ.-ДА/НЕТ;     =
C РЕМОНТ: 0-4 ШИНЫ; СМ. МАСЛА, СМАЗ., РЕГУЛИР.-ДА/НЕТ;              =
C ЗНАЧЕНИЯМ ДА/НЕТ СООТВ. 1 И 0.                                     =
C                                                                    =
C K-ЭЛЕМЕНТ В CELLS, СООТВЕТСТВ. ДАННОЙ МАШИНЕ;                      =
C CELLS-МАССИВ; УСЛУГИ ДЛЯ КАЖДОЙ МАШИНЫ.                            =
C=====================================================================
      INTEGER CELLS(25,6),CARNUM,K
      REAL AMNT(4)
      DATA AMNT/5.,2.,2.,2./,CARNUM/0/
C
C     . . . ВЫЧИСЛИТЬ ОБЪЕМЫ УСЛУГ. . .
      DO 10 J=1,4
         CELLS(K,J) = RAND(AMNT(J))
   10 CONTINUE
      CARNUM = CARNUM+1
      CELLS(K,5) = CARNUM
      RETURN
      END
```

```
      SUBROUTINE FINANS(CAR,L,CELLS,BILL)
C=====================================================================
C ВЫЧИСЛ. СЧЕТОВ ДЛЯ ДЛЯ ОТБЫВ. МАШИН И ОБЩЕЙ ПРИБЫЛИ.               =
C ЕСЛИ CAR Б. ИЛИ Р. 0, НУЖНО ВЫЧИСЛИТЬ СЧЕТ ДЛЯ ДАННОЙ МАШИНЫ.     =
C В СООТВЕТСТВИИ С ЦЕНАМИ:                                           =
C  ЗАПРАВКА: 5 ГАЛ.-3.75; 1КВАРТА МАСЛА-0.85;                       =
C  РЕМОНТ: 1ШИНА-34, СМ.МАСЛА-7.50, СМАЗКА-3.5, РЕГУЛ.-37.95        =
C L=1 И L=2 СООТВЕТСТВУЮТ ЗАПРАВКЕ, L=3-РЕМОНТ.                      =
C СТОИМОСТЬ УСЛУГ ЗАНОСИТСЯ В 'BILL'.                                =
C ЕСЛИ CAR БОЛЬШЕ 0, ТО ЭТО НОМЕР ЭЛЕМЕНТА, СООТВ. ДАННОЙ МАШИНЕ    =
C В CELLS.                                                           =
C ЕСЛИ CAR НЕ БОЛЬШЕ 0, ТО ЭТО ВЗЯТОЕ СО ЗНАКОМ МИНУС ЧИСЛО МАШИН,   =
C УЕХАВШИХ БЕЗ ОБСЛУЖИВАНИЯ.                                         =
C=====================================================================

      INTEGER CELLS(25,6),NSRVD,CAR,PRT
      REAL BILL,PROFIT,COST(4,2),AVE
      DATA NSRVD/0/,PROFIT/0./,COST/3.75,.85,0.,0.,34.,7.5,3.5,37.95/
      DATA PRT/6/
C     . . . СЧЕТ ИЛИ ОБЩАЯ СУММА ПРИБЫЛИ? . . .
      IF(CAR .LE. 0) GO TO 30
C
C     . . . ЗА КАКОЙ ВИД УСЛУГ НУЖЕН СЧЕТ? . . .
      J = 1
      IF(L .EQ. 3) J = 2
C
C     . . . ВЫЧИСЛЕНИЕ СЧЕТА ДЛЯ ДАННОЙ МАШИНЫ . . .
   10 BILL = 0
      DO 20 I=1,4
         BILL = BILL+CELLS(CAR,I)*COST(I,J)
   20 CONTINUE
      PROFIT = PROFIT+.2*BILL
      NSRVD = NSRVD+1
      RETURN
C
C     . . . ПЕЧАТЬ ОБЩЕЙ СУММЫ ПРИБЫЛИ. . .
   30 AVE = PROFIT*5./NSRVD
      WRITE(PRT,101)
  101 FORMAT('0',T23,'**END OF DAY ACCOUNTING**')
      CAR = -CAR
      WRITE(PRT,102) NSRVD,AVE,PROFIT,CAR
  102 FORMAT(' SERVED',I4,' CARS WHO PAID $',F6.2,' ON THE AVERAGE.',
     X   ' TOTAL PROFIT = $',F8.2,/,I4,' CARS TURNED AWAY')
      RETURN
      END
```

Продолжение Рис. 5.23.

```
      INTEGER FUNCTION ARRIVE(TIME,EVENT)
C=====================================================================
C ВЫЧИСЛИТЬ ВРЕМЯ ПРИБЫТИЯ СЛЕДУЮЩЕЙ МАШИНЫ:                         =
C EVENT=4 ОЗНАЧАЕТ ЗАПРАВКУ И РАСПРЕДЕЛЕНИЕ ЭКСПОНЕНЦИАЛЬНОЕ          =
C СО СРЕДНИМ ЗНАЧЕНИЕМ 3 МИН. ДЛЯ ЧАСОВ ПИК                           =
C С 8 ДО 9 И С 4 ДО 6. В ОСТАЛЬНОЕ ВРЕМЯ СРЕДН. ИНТЕРВАЛ              =
C МЕЖДУ ПРИБЫТИЯМИ-6 МИН. (С 9 ДО 4).                                 =
C EVENT=5 ОЗНАЧАЕТ РЕМОНТ, СЛЕДУЮЩЕЕ ПРИБЫТИЕ                         =
C ПЛАНИРУЕТСЯ ЧЕРЕЗ ЧАС.                                               =
C                                                                     =
C TIME -ТЕКУЩЕЕ ВРЕМЯ.                                                 =
C EVENT-ТИП СОБЫТИЯ.                                                  =
C=====================================================================
      INTEGER EVENT,TIME,MU(2),NONPEK(2),GAP,HR,MIN
      DATA MU/3,6/,NONPEK/900,1600/
C
C     . . . ПРИБЫТИЕ НА ЗАПРАВКУ ИЛИ РЕМОНТ? . . .
      IF(EVENT .EQ. 5) GO TO 10
C
C     . . . ЗАПРАВКА: ЧАС ПИК ИЛИ НЕТ? . . .
      J = 1
      IF((TIME .GE. NONPEK(1)) .AND. (TIME .LT. NONPEK(2))) J = 2
      GAP = -MU(J)*ALOG(RAND(1.))
C
C     . . . ВЫЧИСЛ. ВРЕМЯ СЛЕДУЮЩ. ПРИБЫТИЯ. . .
      HR = TIME/100
      MIN = TIME-HR*100+GAP
      HR = HR+MIN/60
      MIN = MIN-(MIN/60)*60
      ARRIVE = HR*100+MIN
      RETURN
C
C     . . . ПРИБЫТИЕ НА РЕМОНТ РОВНО ЧЕРЕЗ ЧАС. . .
   10 HR = TIME/100
      ARRIVE = (HR+1)*100
      RETURN
      END


      INTEGER FUNCTION SHORT(LINK,LANE)
C=====================================================================
C   ВЫЧИСЛИТЬ НОМЕР БОЛЕЕ КОРОТКОЙ ОЧЕРЕДИ.                           =
C                                                                     =
C   LINK  - ПОЛЕ ССЫЛКИ В ЭЛЕМЕНТАХ ОЧЕРЕДИ;                          =
C   LANE  - УКАЗАТЕЛИ НА НАЧАЛА ОЧЕРЕДЕЙ.                             =
C=====================================================================
      INTEGER LINK(25),LANE(3), LANE1, LANE2
C     . . . ПРЕДПОЛОЖИМ, ЧТО ОЧЕРЕДЬ 1 КОРОЧЕ. . .
      SHORT = 1
      LANE1 = LANE(1)
      LANE2 = LANE(2)
C
C     . . . ДВИГАТЬСЯ ПО ОЧЕРЕДЯМ, ПОКА ОДНА ИЗ НИХ НЕ КОНЧИТСЯ . . .
   10 IF(LANE1 .EQ. 0) RETURN
      IF(LANE2 .EQ. 0) GO TO 20
      LANE1 = LINK(LANE1)
      LANE2 = LINK(LANE2)
      GO TO 10
C     . . . ОКАЗАЛОСЬ, ЧТО ОЧЕРЕДЬ 2 КОРОЧЕ. . .
   20 SHORT = 2
      RETURN
      END

      SUBROUTINE OUTPUT(TIME,LANE,CELLS,LINK)
C=====================================================================
C   ТЕКУЩЕЕ СОСТОЯНИЕ ЗАПРАВОЧНОЙ СТАНЦИИ.                            =
C                                                                     =
C   TIME  - ТЕКУЩЕЕ ВРЕМЯ;                                            =
C   LANE  - УКАЗАТЕЛИ НА НАЧАЛА ОЧЕРЕДЕЙ;                             =
C   CELLS - ПО СТРОКЕ НА МАШИНУ, СТОЛБ. 5-НОМЕР МАШИНЫ;               =
C   LINK  - ПОЛЕ ССЫЛКИ В ОЧЕРЕДЯХ.                                   =
C=====================================================================
```

Продолжение Рис. 5.23.

```
      INTEGER TIME,AGAIN,LANE(3),CAR(3),CELLS(25,6),LINK(25),NUM(3),PRT
      LOGICAL MORE
      DATA PRT/6/
C
C     . . . ПЕЧАТЬ ЗАГОЛОВКА ОТЧЕТА. . .
      WRITE(PRT,100) TIME
  100 FORMAT('0',14X,'SERVICE STATION STATUS CHECK:',I5,' HOURS')
      WRITE(PRT,101)
  101 FORMAT(15X, '.........', 10X, '.........', 10X, '.........' /
     1       15X, ': PUMP1 :', 10X, ': PUMP2 :', 10X, ':GARAGE :' /
     2       15X, '.........', 10X, '.........', 10X, '.........' / )
C
C     . . . ПЕЧАТАТЬ СОДЕРЖИМОЕ ВСЕХ ОЧЕРЕДЕЙ. . .
      DO 10 I=1,3
         CAR(I) = LANE(I)
   10 CONTINUE
C
C     . . . НОМЕР МАШИНЫ ИЛИ 0, ЕСЛИ ОЧЕРЕДЬ КОНЧИЛАСЬ. . .
   20 MORE = .FALSE.
      DO 40 I=1,3
         NUM(I) = 0
         IF(CAR(I) .EQ. 0) GO TO 40
         NUM(I) = CELLS(CAR(I),5)
         CAR(I) = LINK(CAR(I))
         MORE = .TRUE.
   40 CONTINUE
      WRITE(PRT,102) (NUM(I),I=1,3)
  102 FORMAT(18X,3(I3,16X))
C     . . . ЕСЛИ НЕ ВСЕ ОЧЕРЕДИ КОНЧИЛИСЬ, ТО НОВАЯ СТРОКА. . .
      IF(MORE) GO TO 20
      WRITE(PRT,103)
  103 FORMAT('0')
      RETURN
      END
```

Продолжение Рис. 5.23.

```
                    состояние заправочной станции: 800 часов

                    .............   .............   .........
                    : колонка 1 :   : колонка 2 :   : ремонт :
                    .............   .............   .........
прибытие в очередь 1 машина номер  1                время 804
прибытие в очередь 2 машина номер  2                время 809
прибытие в очередь 1 машина номер  3                время 810
отбытие из очереди 2 машина номер  2 счет $ 3.75    время 813
отбытие из очереди 1 машина номер  1 счет $12.10    время 815
прибытие в очередь 2 машина номер  4                время 822
прибытие в очередь 1 машина номер  5                время 826
отбытие из очереди 1 машина номер  3 счет $15.85    время 828
отбытие из очереди 2 машина номер  4 счет $11.25    время 829
прибытие в очередь 2 машина номер  6                время 834
прибытие в очередь 1 машина номер  7                время 836
прибытие в очередь 2 машина номер 8                 время 836
отбытие из очереди 1 машина номер  5 счет $15.85    время 840
отбытие из очереди 2 машина номер  6 счет $ 4.60    время 840
отбытие из очереди 2 машина номер  8 счет $ 0.85    время 844
прибытие в очередь 2 машина номер  9                время 845
прибытие в очередь 1 машина номер 10                время 847
прибытие в очередь 2 машина номер 11                время 847
прибытие в очередь 1 машина номер 12                время 848
прибытие в очередь 2 машина номер 13                время 848
отбытие из очереди 1 машина номер  7 счет $11.25    время 849
```

Рис. 5.24. Распечатка программы моделирования.

```
отбытие из очереди 2 машина номер  9 счет $ 7.50 время 849
отбытие из очереди 1 машина номер 10 счет $ 4.60 время 855
отбытие из очереди 2 машина номер 11 счет $ 0.85 время 855
отбытие из очереди 2 машина номер 13 счет $ 0.00 время 858
прибытие в очередь 2 машина номер 14              время 858
отбытие из очереди 1 машина номер 12 счет $ 3.75 время 900
отбытие из очереди 2 машина номер 14 счет $ 0.00 время 900
прибытие в очередь 3 машина номер 15              время 900
```

состояние заправочной станции: 900 ч

колонка 1	колонка 2	ремонт
0	0	15
0	0	0

состояние заправочной станции: 1100 ч

колонка 1	колонка 2	ремонт
32	33	24
34	0	35
0	0	0

```
отбытие из очереди 3 машина номер 143 счет $109.50 время 1747
отбытие из очереди 2 машина номер 140 счет $  7.50 время 1749
отбытие из очереди 1 машина номер 133 счет $  8.35 время 1750
прибытие в очередь 2 машина номер 160              время 1750
прибытие в очередь 1 машина номер 161              время 1753
прибытие в очередь 2 машина номер 162              время 1755
прибытие в очередь 1 машина номер 163              время 1755
** станция переполнена: клиент уехал необслуженным
отбытие из очереди 2 машина номер 142 счет $ 15.00 время 1758
отбытие из очереди 1 машина номер 135 счет $  8.35 время 1800
прибытие в очередь 3 машина номер 164              время 1800
отбытие из очереди 3 машина номер 164 счет $  0.00 время 1800
```

состояние заправочной станции: 1800 часов

колонка 1	колонка 2	ремонт
136	144	0
139	146	0
141	148	0
145	149	0
147	151	0
150	153	0
152	154	0
155	156	0
157	159	0
158	160	0
161	162	0
163	0	0
0	0	0

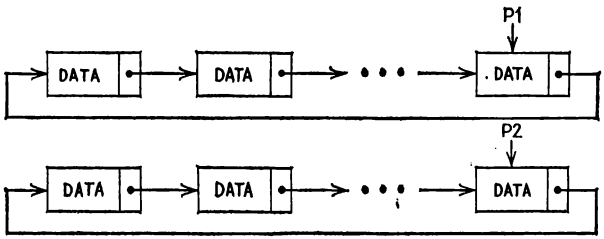
** итог дня **

обслужено 141 клиентов, средний счет 14.47. сумма прибыли = $ 407.99.
4 клиента не были обслужены.

Продолжение Рис. 5.24.

Упражнения

5.01. Предположим, имеется два циклических связанных списка с указателями P1 и P2, как изображено на рисунке.



Пусть P1=0 означает, что первый список пуст, а P2=0 означает, что второй список пуст. Узлы обоих списков состоят из элементов двух целочисленных векторов DATA и LINK. Например, DATA (P1) — первый элемент первого списка, DATA (LINK(P1)) — второй элемент и т. д.

(а) Составьте программу на Фортране, которая подсоединяет второй список в конец первого (конкатенирует их) и делает второй список пустым.

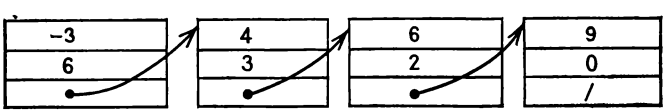
(б) Покажите, как при помощи таких циклических связанных списков можно реализовать операции включения и исключения из очереди. Предполагается, что есть список (стек) свободного пространства с указателем AVAIL. Так, DATA (AVAIL) — первый свободный элемент, DATA (LINK(AVAIL)) — второй и т. д. Значение AVAIL=0 говорит о том, что свободного пространства нет.

5.02. В языке Фортран требуется, чтобы циклы были правильно вложены друг в друга и чтобы конец цикла находился ниже соответствующей инструкции DO. Напишите препроцессор, который бы проверял соблюдение этих условий в программе или подпрограмме на Фортране. (*Указание.* Встретив инструкцию DO, положить на стек метку конца цикла; встретив помеченную инструкцию, взять со стека верхний элемент, если он совпадает с меткой.)

5.03. Многочлен от одной переменной x можно представить в машине как связанный список с узлами вида

Коэффициент при x
Степень x
Ссылка

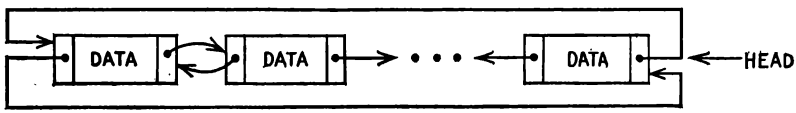
Например, многочлен $-3*x^6+4*x^3+6*x^2+9$ будет храниться в виде списка



в котором узлы расположены в порядке убывания степеней x. Напишите подпрограмму INPUT, которая читает значение n и затем $n+1$ коэффициентов an, . . ., $a1$, $a0$ некоторого многочлена степени n и строит его представление, как указано выше. Узлы с нулевыми коэффициентами включать не нужно. Напишите подпрограмму для сложения таких многочленов (нулевые слагае-

мые должны быть исключены из результирующего списка). Напишите подпрограмму OUTPUT для печати многочлена. Для проверки своих подпрограмм составьте тест, в котором читаются два многочлена, затем складываются и результат печатается.

5.04. Список с двумя связями вида



может быть представлен указателем HEAD и тремя списками RIGHT, LEFT и DATA. Имея такое представление, напишите подпрограммы на Фортране для выполнения следующих операций.
(а) Вернуть указатель на узел со значением поля DATA, равным FIND. Если такой узел не найден, то вернуть признак того, что узел отсутствует.
(б) Исключить элемент, значение поля DATA которого равно DEL.
(в) Добавить узел, на который ссылается указатель PT, в начало списка.
(г) Выполнить операции включения и исключения из очереди.

5.05. Программа сортировки на том же месте, описанная в этой главе, требует n^2 сравнений. Более эффективный способ организации данных для сортировки на том же месте состоит в том, чтобы использовать список NEXT для выборки элементов массива A. Так, если сортировка идет по столбцу COL, то полю NEXT(I) присваивается значение J, если A(J, COL) есть J-й по величине элемент столбца COL. Переменная HEAD при таком методе не нужна. Перепишите программу INSORT, используя алгоритм QUICKSORT из гл. 1, которая бы присваивала соответствующие значения элементам столбца NEXT.

5.06. Подпрограмма TRETAB из этой главы показывает, как найти или добавить элемент в бинарном дереве. Напишите подпрограмму для исключения из бинарного дерева элемента со значением V.

5.07. Вам поручено смоделировать генеалогическое дерево для некоторого лица. Предложите конструкцию такого дерева и постройте алгоритмы для нахождения отца, матери, тети, двоюродного и родного брата этого лица и т. д. Как должно быть устроено такое дерево? Как будет выглядеть его узел?

Глава 6

ПРЕДСТАВЛЕНИЕ ДАННЫХ В МАШИНЕ

Данные всех типов (вещественные, целые, логические, литерные) представляются внутри ЭВМ в виде последовательности двоичных разрядов (цифр), называемых *битами*. На разных машинах приняты разные соглашения о представлении данных. Однако наибольшее распространение получили несколько основных способов кодирования. Вначале мы рассмотрим эти общие принципы, а затем особо остановимся на методах, принятых в машинах серий IBM 360-370 и DEC-10.

6.1. Двоичное представление целых чисел

Двоичная система счисления обладает теми же свойствами, что и более известная десятичная система счисления. Обе они основаны на позиционном принципе. Например, десятичное число 257 представляет собой последовательность из трех цифр: 7 в позиции единиц, 5 в позиции десятков и 2 в позиции сотен. Это можно записать так:

$$257=2*10^2+5*10^1+7*10^0$$

Вообще любое десятичное число может быть представлено в виде последовательности цифр 0, 1, 2, . . . , 9 при основании 10. Позиция каждой цифры указывает степень, в которую нужно возвести 10, чтобы получить множитель при этой цифре.

d_{n-1} d_{n-2} . . . d_2 d_1 d_0 цифра в позиции
10^{n-1} 10^{n-2} . . . 10^2 10^1 10^0 вес позиции

$$\text{Число}=\sum_i 10^i d_i$$

В двоичной системе счисления есть только две цифры 0 и 1. Основание системы 2. Позиция каждой цифры указывает, в какую степень нужно возвести 2, чтобы получить множитель при этой цифре:

b_{n-1} b_{n-2} . . . b_2 b_1 b_0 цифра в позиции
2^{n-1} 2^{n-2} . . . 2^2 2^1 2^0 вес позиции

$$\text{Число}=\sum_i 2^i b_i$$

Таким образом, двоичное число 1101101, например, равно десятичному числу 109, что можно увидеть из следующей записи:

$$1101101 = 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

Хотя в большинстве ЭВМ положительные целые числа представлены в такой двоичной форме (называемой *чисто двоичной*), существуют отличия в способах представления отрицательных чисел. Обычно используются три формы для представления отрицательных чисел: прямой, обратный и дополнительный коды. Рассмотрим каждый из этих кодов. Для простоты во всех приведенных примерах операции выполняются с 10-разрядными двоичными числами.

6.1.1. Прямой код

ЭВМ, в которой для представления двоичных чисел используется *прямой* код, запоминает числа в чисто двоичном коде. В этом коде один бит числа отводится для представления знака. Например, десятичное число 14 в прямом коде будет записано так:

0000001110

Крайний левый двоичный разряд 0 указывает на знак плюс (+), остальная цепочка битов 000001110 представляет десятичное число 14. Десятичное число —17 хранится в машине как

1000010001

Здесь крайний левый бит 1 обозначает знак минус (—), а биты 000010001 представляют десятичное число 17.

Представление чисел в прямом коде удобно для интерпретации человеком, но имеет два недостатка. Первый, у числа 0 нет однозначного представления, так как оба числа 0000000000 (+0) и 1000000000 (—0) представляют ноль. Второе и более важное обстоятельство — возникают трудности при выполнении в прямом коде операций сложения и вычитания. Они вызваны тем, что при сложении чисел с разными знаками должны приниматься во внимание относительные величины чисел. Например, при сложении чисел 14 и 17 мы не можем просто сложить их двоичные представления, поскольку в этом случае получили бы неверный результат:

$$\begin{array}{rr} 0000001110 & +14 \\ +1000010001 & -17 \\ \hline 1000011111 & -31 \end{array}$$

Поэтому, если знаки чисел различны, мы должны вычесть меньшую величину из большей и затем приписать правильный знак:

```
 000010001     +17
-000001110     -(+14)
----------     ------
1000000011       -3
```

Этот недостаток привел к тому, что в большинстве вычислительных машин были приняты другие способы кодирования.

6.1.2. Обратный код

Представление отрицательных чисел в *обратном* коде принято на многих машинах, в том числе серий UNIVAC 1100 и CDC 6000 и 7000. В обратном коде крайний левый бит также представляет знак, но в арифметической операции участвуют все двоичные разряды, включая бит знака.

Десятичные числа +14 и —17 будут представлены в обратном коде как

$$0000001110 = +14$$
$$1111101110 = -17$$

Число —17 вначале представляется как +17 (0000010001), а затем все 1 заменяются на 0, а 0 — на 1. Для представления числа $-x$ в n-разрядном двоичном слове достаточно вычесть x из последовательности, содержащей n единиц. Таким образом, обратный код числа x есть 2^n-1-x.

Теперь при выполнении операции сложения —17 и +14 мы имеем

```
 0000001110     +14
+1111101110     -17
----------     ----
 1111111100      -3
```

Однако правильный ответ не всегда так легко получается. Рассмотрим, например, суммирование чисел 17 и —14:

```
 0000010001     +17
+1111110001     -14
----------     ----
 0000000010      +2
```

Результат оказался на единицу меньше истинного. Добавив единицу переноса из крайнего левого разряда суммы, получим правильный ответ:

```
 0000010001     +17
+1111110001     -14
----------     ----
 0000000010      +2
         1      +1 (бит переноса)
----------     ----
 0000000011      +3
```

Вообще при сложении чисел в обратном коде вначале выполняется суммирование и затем, если возникает перенос из знакового разряда, добавляется единица к полученному результату. Таким образом, при работе с обратным кодом первое затруднение состоит в необходимости дополнительной операции сложения. Второе затруднение заключается в том, что число ноль не имеет единственного представления: оба представления 0000000000 (+0) и 1111111111 (—0) допустимы. Результат —0 получается при вычитании отличного от нуля числа x из самого себя. Например,

```
  0000000101     +5
+ 1111111010     -5
  ----------     --
  1111111111     -0
```

6.1.3. Дополнительный код

Число $-x$ в n-разрядном *дополнительном* двоичном коде представляется в виде 2^n-x. Дополнительный код может быть получен из обратного кода числа x (2^n-1-x) прибавлением 1 к младшему разряду ($2^n-1-x+1=2^n-x$). Так, число —17 в дополнительном коде будет записано как

$$1111101111=-17$$

Процедура изменения знака — это основной недостаток дополнительного кода. В отличие от обратного и прямого кода для изменения знака числа, представленного в дополнительном коде, требуется выполнить операцию сложения. Положительной стороной дополнительного кода является то, что число 0 имеет единственное представление. Вот как в дополнительном коде представляется число ноль:

```
  1111111111     (обратный код)
+            1    (прибавление 1)
  ----------     (бит переноса игнорируется)
  0000000000
```

Одно из наиболее важных достоинств дополнительного кода состоит в том, что при сложении не возникает особых случаев. В качестве примера проиллюстрируем сложение чисел +14 и —17:

```
  0000001110     +14
+ 1111101111     -17
  ----------     ---
  1111111101      -3
```

6.2. Двоичное представление вещественных чисел

В десятичной системе счисления число 12.63 представляется в виде суммы

$$12.63=1*10+2*1+6*.1+3*.01$$

Аналогично двоичное число 110.11 будет представлено как

$$110.11 = 1*4 + 1*2 + 0*1 + 1*.5 + 1*.25$$

Такая интерпретация числа основана на том, что в позиционной системе счисления цифра, расположенная правее разделительной (или позиционной) [1]) точки, должна быть умножена на основание системы счисления в соответствующей отрицательной степени.

Для представления вещественных чисел в ЭВМ нужно каким-то образом обозначить положение десятичной точки. В большинстве случаев используется представление, называемое *числом с плавающей точкой*. Оно основано на так называемой научной нотации. Десятичное число 12.63, например, будет представлено в виде $.1263*10^2$, а двоичное число 110.11 — как $.11011*2^3$.

В обоих случаях число состоит из двух частей — мантиссы (или дробной части) и показателя. Дробная часть может быть *нормализована*. Нормализованная положительная мантисса — это мантисса, у которой первая после точки цифра отлична от нуля (цифра может быть нулем только в том случае, если вся мантисса равна нулю). Достоинством такой формы записи является то, что любое число имеет единственное представление. Для ненормализованного же числа существует сколь угодно много различных представлений (например, число 12.63 может быть записано как $.01263*10^3$, $126.3*10^{-1}$ и т. п.).

Если для представления чисел используется описанный способ, то они хранятся в памяти обычно в следующем виде:

Знак	Порядок	Мантисса

Порядок представляет показатель степени. Так, двоичное число 100.11 в 10-разрядном двоичном слове будет записано в виде

0	1011	10011
Знак	Порядок +8	Дробная часть

В данном примере мы отвели четыре двоичных разряда под порядок и пять разрядов под мантиссу. Порядок записан в форме «плюс восемь». Это означает, что указанное значение показателя на 8 больше истинного. Так, двоичный порядок 1000 соответствует нулевому показателю, 0=8—8, 1011 — показателю 3=11—8, 0011 — показателю —5=3—8 и т. д.

---

[1]) В десятичной системе счисления ее называют еще и десятичной точкой.— *Прим. ред.*

Может показаться странным, почему мы используем форму «плюс восемь» вместо представления в прямом, обратном или дополнительном коде. На это есть две причины. Первая — легче производить сравнение абсолютных величин: порядок и мантиссу можно обрабатывать как одну последовательность битов. Вторая причина — упрощается сложение, так как вначале мы сравниваем показатели чисел, а затем сдвигаем вправо мантиссу с меньшим показателем до тех пор, пока порядки чисел не совпадут. Так, при сложении чисел 110.11 (6.75) и .01 (.25) выполняются следующие действия:

1. Определение большего порядка:

0:1 0 1 1:1 1 0 1 1	(6.75)	$.11011 * 2^3$
0:0 1 1 1:1 0 0 0 0	(0.25)	$.10000 * 2^{-1}$
Разность = 4		

2. Выравнивание порядков и последующее сложение:

0:1 0 1 1:1 1 0 1 1	(6.75)	$.11011 * 2^3$
+0:1 0 1 1:0 0 0 0 1	(0.25)	$.00001 * 2^3$
0:1 0 1 1:1 1 1 0 0	(7.00)	$.11100 * 2^3$

Описанная методика не отличается от той, которую мы применяем при сложении десятичных чисел. Например, если бы мы складывали числа $.675*10^1$ и $.25*10^0$, то вначале выравнивали бы показатели путем замены числа $.25*10^0$ на $.025*10^1$, а затем произвели бы сложение мантисс.

При сложении двух чисел результирующее значение мантиссы может получиться ненормализованным. В этом случае необходимо выполнить последующую нормализацию *(постнормализацию)*. Следующие два примера иллюстрируют необходимость выполнения постнормализации.

Случай 1. Мантисса результата слишком велика.

1. Определение большего порядка:

0:1 0 1 1:1 1 0 1 1	(6.75)	$.11011 * 2^3$
0:1 0 0 1:1 1 1 0 0	(1.75)	$.11100 * 2^1$
Разность = 2		

2. Выравнивание порядков и сложение:

0:1 0 1 1:1 1 0 1 1	(6.75)	$.11011 * 2^3$
+0:1 0 1 1:0 0 1 1 1	(1.75)	$.00111 * 2^3$
0:1 1 0 0:1 0 0 0 1	(8.50)	$1.00010 * 2^3 = .10001 * 2^4$

Вычисленное значение мантиссы равно 1.00010. Для нормализации полученного результата прибавляем 1 к порядку, заменяя 1011 на 1100, и делим мантиссу на два (сдвигая ее вправо на один разряд).

Случай 2. Мантисса результата слишком мала.

1. Определение большего порядка:

```
0:1 0 1 1 :1 1 0 1 1   (6.75)    .11011 * 2³
1:1 0 1 0 :1 1 0 0 0   (—3.0)    .11000 * 2²
Разность = 1
```

2. Выравнивание порядков и сложение:

```
 0:1 0 1 1:1 1 0 1 1   (6.75)     .11011 * 2²
+1:1 0 1 1:0 1 1 0 0   (—3.0)    —.01100 * 2³
 0:1 0 1 0:1 1 1 1 0   (3.75)     .01111 * 2³ = .11110 * 2²
```

Выше мы полагали, что отрицательные числа представляются в прямом коде. Затруднение в данном случае возникло, когда мы получили дробную часть .01111. Была выполнена нормализация за счет вычитания из порядка 1 и умножения мантиссы на 2 (сдвигом на один разряд влево).

Отрицательные вещественные числа могут быть представлены в прямом, обратном или дополнительном коде. Число ноль, однако, всегда представляется как правильный ноль (все биты нулевые). Такое представление гарантирует, что ноль имеет наименьший порядок. Поэтому, если ноль прибавляется к другому числу, не должна происходить потеря битов в дробной части этого числа при переносе разделительной точки.

6.3. Ошибки и потеря точности — целые числа

Во всех рассмотренных нами примерах арифметики с целыми числами мы могли получить правильный результат. Однако в действительности может получиться неверный или неточный результат. Наиболее общим случаем неверного результата является *переполнение*: при сложении или умножении двух чисел результат не умещается в одно слово памяти машины. Например, при сложении чисел 71 и 445 в ЭВМ, где числа представлены 10-разрядным двоичным дополнительным кодом:

```
 0001000111      71
+0110111101     445
 1000000100    —508 (ошибка переполнения)
```

в результате ошибки переполнения получается неверный результат —508. Обычно ошибки такого типа ЭВМ обнаруживает, прекращает выполнение программы и выдает соответствующее сообщение об ошибке. К сожалению, не во всех машинах предусмотрена регистрация таких ошибок.

Второй наиболее общий случай возникновения ошибок — это *деление на ноль*. Практически все машины обнаруживают эту

ошибку, выполнение программы прекращается и выдается соответствующее сообщение.

Даже когда при делении целых чисел частное не равно 0, полученный результат часто бывает очень неточным. Рассмотрим, например, вычисление целых выражений

$$I+J/K*L$$

и

$$I+J*L/K$$

Математически они идентичны. При I=3, J=4, K=5 и L=6 правильный результат равен 7.8, однако при вычислении первого выражения будет получен результат 3, а при вычислении второго 7. Действительно,

$$I+J/K*L=3+(4/5)*6=3+0*6=3+0=3$$
$$I+J*L/K=3+(4*6)/5=3+24/5=3+4=7$$

Как видно из данного примера, порядок вычислений имеет важнейшее значение. Обычно выполнение операции деления следует откладывать как можно дольше, чтобы избежать кумулятивного эффекта — накопления и распространения ошибок усечения.

6.4. Ошибки и потеря точности — вещественные числа

Наиболее типичные ошибки, связанные с арифметикой над вещественными числами, возникают в трех случаях: *переполнение порядка* (величина числа слишком велика), *потеря порядка* (величина отличного от нуля числа слишком мала) и *деление на ноль*. Как и в случае возникновения ошибок в арифметике с целыми числами, большинство ЭВМ обнаруживает указанные ошибки в арифметике с вещественными числами, в результате чего выполнение программы прекращается и выдается сообщение об ошибке.

К более серьезным ошибкам, которые машина не обнаруживает, относятся ошибки неточности вычислений. Неточность полученных результатов может возникать по разным причинам. Во-первых, вследствие того, что мантисса имеет конечную длину, некоторые числа невозможно точно представить. Так как дробь 1/3, например, не может быть точно представлена в десятичной системе счисления, то десятичное число .3 не может быть точно представлено двоичной мантиссой конечной длины. Поэтому при выполнении присваивания

X=.3

вносится небольшая ошибка в результаты, полученные программой.

Арифметические операции, такие, как сложение и умножение, вносят дополнительные ошибки из-за округлений или усечений.

необходимых при запоминании слишком большой дробной части в машинном слове, имеющем конечную длину. Алгоритмы, позволяющие сократить число арифметических операций, обычно снижают эффект накопления ошибки.

Ошибки округления и усечения зависят также от порядка выполнения операций. Например, число 64 представлено в 10-разрядном двоичном коде как

$$0{:}1111{:}10000 \quad +64 \quad .10000*2^7$$

а число 1 — как

$$0{:}1001{:}10000 \quad +1 \quad .10000*2^1$$

Теперь, вычисляя (64—64)+1, мы получим результат 1. Однако если порядок вычисления будет иным, (64+1)—64, то мы получим неверный результат 0. Чтобы убедиться в этом, покажем, что в результате сложения 64+1 получается 64.

1. Определяем больший порядок:

$$\begin{array}{lll} 0{:}1\ 1\ 1\ 1{:}1\ 0\ 0\ 0\ 0 & +64 & .10\,000*2^7 \\ 0{:}1\ 0\ 0\ 1{:}1\ 0\ 0\ 0\ 0 & +1 & .10\,000*2^1 \\ \hline \text{Разность} = 6 & & \end{array}$$

2. Выравниваем порядок, сдвигая мантиссу:

$$\begin{array}{lll} 0{:}1\ 1\ 1\ 1{:}1\ 0\ 0\ 0\ 0 & +64 & .10\,000\ \ *2^7 \\ +0{:}1\ 1\ 1\ 1{:}0\ 0\ 0\ 0\ 0\ 0\ 1 & +1 & .000\,0001*2^7 \\ \hline 0{:}1\ 1\ 1\ 1{:}1\ 0\ 0\ 0\ 0 & +64 & .10\,000\ \ *2^7 \end{array}$$

Проблема, которая возникает в данном случае, вызвана большой разницей в величинах складываемых чисел (слишком мал диапазон чисел, представимых в 10-разрядном двоичном слове). На практике длина слова обычно намного больше, поэтому число 64 не будет казаться слишком большим по сравнению с 1. Однако в большинстве машин (в которых отводится не более 35 битов для представления мантиссы) вычисление выражения

$$(2{**}35+2{**}-35)-2{**}35$$

дает ноль, тогда как вычисление

$$(2{**}35-2{**}35)+2{**}-35$$

приводит к правильному результату 2^{-35};

Как можно избежать потери точности при вычислениях, показывает решение квадратного уравнения вида

$$ax^2+bx+c=0.$$

Одно из решений этого уравнения

$$(-b+\sqrt{(b^2-4ac)})/2a.$$

Если $a=1$, $b=10^5$ и $c=-1$, получим

$$(-10^5+\sqrt{(10^{10}+4)})/2.$$

На вычислительных машинах, в которых арифметические операции выполняются с точностью менее 10 десятичных разрядов (например, IBM 360-370 и DEC-10), $10^{10}+4$ дает 10^{10}, $\sqrt{(10^{10})}$ дает 10^5 и $(-10^5+10^5)/2$ даст в результате значение x, равное 0. Это абсолютно неверно, так как

$$1*0^2+10^5*0-1=-1.$$

В том случае, если b^2 много больше $4ac$, лучшее приближение получится при умножении

$$(-b+\sqrt{(b^2-4ac)})/2a$$

на тождество

$$(-b-\sqrt{(b^2-4ac)})/(-b-\sqrt{(b^2-4ac)}).$$

Это дает эквивалентную формулу

$$2c/(-b-\sqrt{(b^2-4ac)}).$$

Хотя полученная формула алгебраически эквивалентна первоначальной, при ее применении происходит вычисление

$$2/(10^5+\sqrt{(10^{10}+4)}).$$

Полагая, что точность менее 10 цифр, получим $x=10^{-5}$. Хотя этот результат неточен, он намного лучше прежнего, в чем можно легко убедиться:

$$1*(10^{-5})^2+10^5*10^{-5}-1=10^{-10}.$$

Более подробное обсуждение предмета точности вычислений выходит за рамки данной книги. Читатели, интересующиеся этой проблемой, могут найти подробную информацию в книгах по численным методам, применяемым в ЭВМ [1]).

6.5. Двойная точность

Число разрядов, определяющих точность вычислений, проводимых над вещественными переменными, недостаточно для многих применений. Поэтому Фортран обеспечивает тип данных, называемый «переменная с двойной точностью». Переменные с двойной точностью декларируются посредством инструкции *DOUBLE PRECISION:*

DOUBLE PRECISION *список переменных*

[1]) См., например, книгу Д. Кнута, Искусство программирования для ЭВМ. Т. 2.— М.: Мир, 1977, п.4.2.2. Там же читатель найдет материал, относящийся к другим разделам этой главы.— *Прим. ред.*

Константа с двойной точностью будет записана точно так же, как и вещественная константа Е-типа, только вместо буквы Е следует написать букву D. Например,

2.62144000976525 D+5

представляет число 262144.000976525. Различные вычислительные машины имеют различную точность представления данных вещественного типа и с двойной точностью, но всегда представление с двойной точностью дает по крайней мере вдвое большую точность по сравнению с обычной арифметикой вещественных чисел.

Переменные с двойной точностью обычно считываются и выводятся на печать в формате D. Спецификацию D записывают аналогично Е-формату с той разницей, что буква Е заменяется на D. Код формата D интерпретируется идентично Е. На самом деле эти коды могут быть использованы равнозначно.

Общеприняты два способа хранения чисел двойной точности в памяти ЭВМ. В первом для запоминания используют два слова, каждое с порядком и мантиссой. Мантисса первого слова содержит старшие разряды дробной части, младшие разряды хранятся во втором слове. Представленное таким образом число определяется суммой двух слов. Так, если мантисса состоит из n цифр, порядок второго слова на n меньше порядка первого слова. Применительно к 10-битным словам можно представить десятичное число 8.875 (1000.111 в двоичной системе счисления) как

0 : 1100 : 10001 0 : 0111 : 11000

Отсюда видно, что первая часть представляет десятичное число 8.5, а вторая .375; сумма этих чисел равна 8.875.

Этот первый способ вносит некоторую избыточность, поскольку порядок второй части может быть вычислен по порядку первого слова. Во втором способе представления в некоторых машинах используются все биты второго слова (или все биты, кроме знакового) для расширения мантиссы первого слова. В таких машинах число 8.875 может быть представлено как

0 : 1100 : 10001 1100000000

или, если знаковый бит второго слова не использован, как

0 : 1100 : 10001 0 : 110000000

6.6. Восьмеричные и шестнадцатеричные числа

Во всех вышеприведенных примерах данной главы мы ограничились словами с 10 двоичными разрядами. Такое представление чисел удобно для объяснений, но нереалистично в практических применениях. Современные машины, такие, как DEC-10

и IBM 360-370, имеют гораздо больший размер слова: 36 и 32 двоичных разряда соответственно. Для более компактного представления слов такой длины обычно выражают содержимое слова DEC-10 в *восьмеричной системе счисления* (с основанием 8), а слова IBM 360-370 в *шестнадцатеричной системе счисления* (с основанием 16).

В то время как десятичные числа представляются цепочкой цифр от 0 до 9, восьмеричные числа представляет цепочка цифр от 0 до 7. В десятичном числе

$$d_{n-1}\ d_{n-2}\ \ldots\ d_2\ d_1\ d_0$$

с позицией i (цифрой d_i) связан вес 10^i. Аналогично восьмеричное число

$$o_{n-1}\ o_{n-2}\ \ldots\ o_2\ o_1\ o_0$$

имеет вес 8^i, соответствующий позиции i. Величина такого числа:

$$\text{Число} = \sum_i 8^i\ o_i.$$

Так, восьмеричное число 324 можно записать в виде

$$3*8^2+2*8^1+4*8^0,$$

что дает 192+16+4, или 212. Так как $8=2^3$, каждая цифра в восьмеричной записи соответствует трем двоичным цифрам:

Восьмеричные числа	Двоичные числа
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Шестнадцатеричные	Двоичные	Десятичные
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Так, восьмеричное число 324 соответствует двоичному числу 011010100. Запомнив приведенную выше таблицу, вы легко сможете преобразовывать числа из восьмеричной системы счисления в двоичную и обратно.

В шестнадцатеричной системе счисления используются цифры от 0 до 9 и от A до F; основание системы 16. Учитывая, что $16=2^4$, можно установить соответствие между шестнадцатеричными цифрами и их двоичным и десятичным представлением — см. стр. 276. Используя эту таблицу, можно записать двоичное число 101001101111 как A6F в шестнадцатеричной системе счисления или, суммируя

$$10*16^2+6*16^1+15*16^0,$$

как 2671 в десятичной системе счисления.

6.7. Представление чисел в IBM 360-370

В IBM 360-370 целые и вещественные числа запоминаются в слове с 32 двоичными разрядами (восемь шестнадцатеричных цифр). Целые числа представлены в машине в дополнительном коде. Переполнение, если оно происходит, будет обнаружено в результате выполнения операций сложения или вычитания. Регистрируется также деление на ноль. К сожалению, если произведение двух целых чисел нельзя представить в 32 битах, ошибка не будет обнаружена. Поэтому ошибочное произведение останется незамеченным и может испортить все последующие вычисления.

Вещественные числа в IBM 360-370 представлены в прямом коде и хранятся нормализованными в следующем формате:

Знак	Порядок	Мантисса
0	1—7	8—31

Порядок превышает на 64 истинный шестнадцатеричный (не двоичный) показатель. Так, десятичные числа 25.75 и —25.75 хранятся в шестнадцатеричном виде как

42 : 19C000 25.75

и

C2 : 19C000 —25.75

Шестнадцатеричный порядок имеет свои достоинства и недостатки. Положительной стороной является то, что, используя только семь битов, мы можем представить числа в диапазоне от 16^{-64} до 16^{63}. Основной недостаток — нормализованная шестнадцатеричная дробная часть может оказаться ненормализо-

ванной, если ее рассматривать как двоичную. Например, дробная часть числа 25.75 в двоичном представлении IBM 360-370 есть

.000110011100000000000000

В среднем машины IBM 360-370 обеспечивают точность лишь в пределах 22.5 двоичных разрядов, а не 24, как это может показаться.

Числа с двойной точностью имеют длину 64 бита, в составе которых 32 дополнительных бита используются для расширения мантиссы. Эти числа записывают в форме

Знак	Порядок	Мантисса
0	1—7	8—63

Так, например, десятичное число $262144.0009765625 = 2^{18} + 2^{-10}$ хранится в шестнадцатеричной системе счисления как

45 : 400000 04000000

В среднем «двойная точность» означает 54.5 двоичных разрядов. В десятичной записи точность увеличивается с 7 до 16 десятичных цифр. Диапазон представимых чисел в обоих случаях составляет примерно от 10^{-78} до 10^{75} (от 16^{-65} до 16^{63}).

Из-за относительно невысокой точности в машинах IBM 360-370 возникает необходимость в *разряде защиты*. Это дополнительная шестнадцатеричная цифра используется во всех операциях над вещественными числами и над числами с двойной точностью. Проиллюстрируем ее роль в наглядной форме.

Пример: Вычесть 41112217 из 4210778A.

Чтобы выровнять порядки, преобразуем 41112217 в 420112217.

Дополнительная цифра мантиссы (7) используется как разряд защиты:

```
 4 2:1 0 7 7 8 A  0 ←  разряд защиты
-4 2:0 1 1 2 2 1  7 ←  разряд защиты
-----------------  -
 4 2:0 F 6 5 6 8  9 ←  разряд защиты
 4 1:F 6 5 6 8 9       Нормализация
```

Польза разряда защиты становится очевидной при постнормализации. Если бы разряда защиты не было, результат имел бы меньшую точность:

4 1 : F 6 5 6 9 0

При выполнении арифметических операций над вещественными числами и числами с двойной точностью машины IBM 360-370 обнаруживают ошибки переполнения и потери порядка, а также ошибки, возникающие при делении на ноль.

6.8. Организация памяти в машинах IBM 360-370

6.8.1. Байты, полуслова, слова и двойные слова

Основная единица памяти в машинах IBM 360-370 имеет длину 8 бит и называется *байтом*. Каждый байт имеет свой адрес, с помощью которого к нему осуществляется доступ. Первый байт памяти имеет адрес ноль, второй — единица и т. д.

Байты группируются для формирования более крупных ячеек памяти. Любая пара последовательных байтов, в которой первый байт имеет четный адрес, называется *полусловом*. Так, нулевой и первый байты вместе образуют полуслово. Аналогично полуслово образуют байты второй и третий, а байты первый и второй полуслово не образуют.

Слово состоит из четырех последовательных байтов, где первый байт имеет адрес, кратный 4. *Двойное слово* представлено восемью последовательными байтами, первый из которых имеет адрес, кратный 8. Следующая схема показывает связь полуслов (H), слов (F) и двойных слов (D) с первыми 16 байтами памяти ЭВМ:

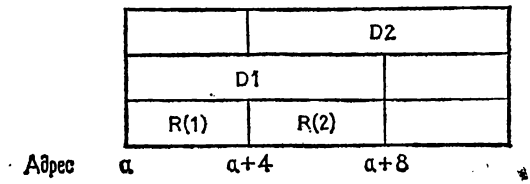
	D				D			
	F		F		F		F	
	H	H	H	H	H	H	H	H
Адрес	00′ 01′	02′ 03′	04′ 05′	06′ 07′	08′ 09′	10′ 11′	12′ 13′	14′ 15′

Машины серии 360 построены таким образом, что операции над целыми, вещественными и логическими переменными могут быть выполнены только в том случае, если под указанные переменные отведено полное слово. Переменные с двойной точностью должны располагаться обязательно в двойных словах. Хотя на IBM 370 такого ограничения нет, эффективность вычислений резко снижается, если элементы данных размещены иначе. По этим причинам все компиляторы IBM 360-370 пытаются отводить целым, вещественным и логическим переменным поле длиной в слово, а переменным с двойной точностью — двойное слово.

При программировании на Фортране необходимо знать соглашения о словах и двойных словах, поскольку незнание этих соглашений может помешать компилятору правильно распределить память под переменные, если мы пользуемся инструкциями COMMON и EQUIVALENCE. Рассмотрим в качестве примера следующие декларации:

```
REAL R(2)
DOUBLE PRECISION D1, D2
EQUIVALENCE (D1,R(1)), (D2,R(2))
```

Этот набор деклараций требует, чтобы память была распределена следующим образом:



Но если для D1 отведено двойное слово, то D2 не попадет на границу двойного слова, и наоборот. На машинах серии 360 такое распределение привело бы к неисправимой ошибке (нарушение границ), а на IBM 370 это привело бы к существенному увеличению времени вычислений.

Не столь очевидное затруднение возникает при выполнении следующей последовательности деклараций:

```
REAL R
DOUBLE PRECISION D
EQUIVALENCE (R,D)
```

Может показаться, что выполнить такое распределение не составляет труда, поскольку любой компилятор должен отвести для D двойное слово, а потому и R, и D будут расположены правильно. К сожалению, большинство компиляторов при определении границ «смотрят» только на первую переменную в EQUIVALENCE. Поэтому пара (R, D) интерпретируется так, что переменные желательно выравнять только на границу слова. В противоположность этому

```
EQUIVALENCE (D, R)
```

предполагает выравнивание на границу двойного слова.

Задавая эквивалентность литерных переменных и переменных других типов, литерные переменные в EQUIVALENCE следует записывать последними, поскольку для этого типа данных никаких требований на установление границ не существует.

Проблема при использовании инструкции COMMON возникает в связи с тем, что всякая общая область начинается с границы двойного слова. Поэтому, если D — переменная с двойной точностью, а R — вещественная переменная, то инструкция

```
COMMON D, R
```

обеспечивает правильное выравнивание, а инструкция

```
COMMON R, D
```

неправильное. Эти потенциальные затруднения всегда могут быть устранены за счет такой организации общей области, при

которой переменные с двойной точностью располагаются перед вещественными, целыми или логическими переменными. Литерные переменные следует располагать в конце общей области.

6.8.2. Типы переменных

Ранее мы познакомились с пятью типами переменных Фортрана: DOUBLE PRECISION, REAL, INTEGER, LOGICAL и CHARACTER*n*. Кроме того, в некоторых диалектах Фортрана для IBM 360-370 допускаются переменные типов REAL*8, REAL*4, INTEGER*4, INTEGER*2, LOGICAL*4 и LOGICAL*1. Цифры в этих декларациях определяют число байтов памяти, которое отводится для переменной. Так, декларация REAL*8 эквивалентна DOUBLE PRECISION; REAL 4 эквивалентна REAL; INTEGER*4 — INTEGER; LOGICAL*4 — LOGICAL. Лишь INTEGER*2 и LOGICAL*1 можно считать расширением того языка, которым мы занимались до сих пор.

Переменные типа INTEGER*2 являются двухбайтовыми целыми и хранятся в памяти в адресуемых полусловах. Этот тип переменных следует использовать в том случае, если предполагается, что целые значения по абсолютной величине будут не больше, чем 32 767, т. е. (2^{15}—1). Если необходим большой массив целых чисел, применение декларации INTEGER*2 может дать существенную экономию памяти. Но эта декларация не лишена и некоторых недостатков. Основной из них — нестандартность данного типа переменной. Его использование может ограничить портативность программ.

В некоторых моделях IBM 360 существует проблема эффективности, связанная с применением INTEGER*2. Для выполнения арифметических операций с полусловами может потребоваться машинного времени больше, чем при работе со словами. Например, на IBM 360 модели 65 операция сложения полуслов занимает на 28% машинного времени больше, чем та же операция со словами. Эта очевидная аномалия объяснима: машина модели 360/65 приспособлена для выполнения операции сложения только со словами. Сложение полуслов выполняется преобразованием полуслов в эквивалентные слова (путем распространения знакового разряда на два байта влево), затем происходит сложение образованных таким образом слов и, наконец, обратное преобразование результата в полуслово. Если для результата требуется более 16 битов, то усеченный результат оказывается неверным. К сожалению, аппаратные средства в машинах IBM 360-370 не обнаруживают таких ошибок переполнения полуслова.

В свете вышеприведенных доводов за и против декларацией INTEGER*2 следует пользоваться разумно. Например, часто используемые счетчики никогда не должны быть типа INTEGER*2.

Если переменные типа INTEGER*2 включены в инструкции EQUIVALENCE или COMMON, они должны быть расположены после переменных двойной точности, вещественных, целых и логических переменных, но перед литерными переменными. Такое расположение переменных гарантирует, что этим целым переменным будут отведены полуслова.

6.8.3. Переменные типа LOGICAL*1 и действия с литерами

Поскольку логические переменные могут принимать только значения .TRUE. и .FALSE. , кажется излишним отводить им поля длиной в слово (32 двоичных разряда). Идеально было бы использовать только один бит для каждой такой переменной. К сожалению, память поразрядно не адресуема, и поэтому лучшее, на что можно рассчитывать, это использовать один байт. Именно это и происходит, когда мы объявляем логическую переменную LOGICAL*1. Как и в случае с INTEGER*2-переменной, применение переменной LOGICAL*1 может сэкономить память, но нарушит мобильность программы.

В основном переменные типа LOGICAL*1 применяют для работы с литерами, когда в компиляторе переменные литерного типа не предусмотрены. В связи с тем что для хранения литеры требуется один байт, память можно существенно сэкономить, если цепочки литер запоминать в массивах типа LOGICAL*1, а не в целых, вещественных или логических массивах. Однако это преимущество несколько уменьшается по той причине, что нельзя воспользоваться операцией сравнения для переменных типа LOGICAL*1. Для обхода этого препятствия программисты часто пишут короткие подпрограммы, в которых сравнение LOGICAL*1-переменных выполняется косвенно. Иллюстрируемая здесь функция CHEQ — одна из таких подпрограмм. Она возвращает значение .TRUE. в том случае, если переменные L1 и L2 типа LOGICAL*1 содержат идентичные значения.

```
      LOGICAL FUNCTION CHEQ (L1, L2)
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C * СРАВНЕНИЕ ОДИНОЧНЫХ ЛИТЕР, ЯВЛЯЮЩИХСЯ ЗНАЧЕ-  *
C *        НИЯМИ ПАРАМЕТРОВ L1, L2 ТИПА LOGICAL*1.                   *
C * ИСПОЛЬЗУЕМЫЙ МЕТОД —L1 и L2 ПЕРЕСЫЛАЮТСЯ В КРАЙ-  *
C *        НИЕ ПРАВЫЕ БАЙТЫ ЦЕЛЫХ ПЕРЕМЕННЫХ L1VAL И  *
C *        L2VAL СООТВЕТСТВЕННО. ФУНКЦИЯ ВОЗВРАЩАЕТ  *
C *        РЕЗУЛЬТАТ СРАВНЕНИЯ ЭТИХ ДВУХ ПЕРЕМЕННЫХ.  *
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      LOGICAL*1 L1, L2, L1EXT(4), L2EXT(4)
      INTEGER L1VAL, L2VAL
      EQUIVALENCE (L1VAL, L1EXT(1)), (L2VAL, L2EXT(1))
C
C       ЗАНУЛИТЬ L1VAL и L2VAL. ПРИСВОИТЬ ЗНАЧЕНИЯ L1
C       И L2 КРАЙНИМ ПРАВЫМ БАЙТАМ.
```

```
C       ПРИСВОИТЬ ПЕРЕМЕННОЙ CHEQ ЗНАЧЕНИЕ .TRUE., ЕСЛИ
C       L1VAL=L2VAL.
        L1VAL=0
        L2VAL=0
        L1EXT(4)=L1
        L2EXT(4)=L2
        CHEQ= (L1VAL .EQ. L2VAL)
        RETURN
        END
```

6.9. Представление чисел в ЭВМ DEC-10

В ЭВМ DEC-10 целые и вещественные числа хранятся в словах длиной 36 двоичных разрядов (12 восьмеричных цифр). Целые числа представлены в машине в дополнительном коде. Деление на ноль, ошибки переполнения в операциях над целыми и вещественными числами, включая те, что происходят в результате умножения целых чисел, обнаруживаются машиной.

Вещественные числа запоминаются в нормализованном виде; используется формат

Знак	Порядок	Мантисса
0	1—8	9—35

Отрицательные вещественные числа представлены в дополнительном коде. Порядок увеличен на 128 по сравнению с двоичным показателем. Так, например, десятичные числа 25.75 и —25.75 хранятся (в восьмеричной системе счисления) в виде

205 : 634000000

и

572 : 144000000

соответственно. Заметьте, что отрицательное число —25.75 образовано по правилам дополнительного кода во всех двоичных разрядах, включая порядок, по отношению к представлению числа 25.75.

Структура представления чисел с двойной точностью в старых моделях (KA-процессоры) DEC-10 отличается от более современных моделей (KI- и KL-процессоры). На KA-процессоре числа с двойной точностью хранятся в виде

Знак	Порядок	Мантисса 1	Знак	Порядок—27	Мантисса 2
0	1—8	9—35	36	37—44	45—71

Так, например, десятичное число $262144.0009765625 = 2^{18} + 2^{-10}$ (в восьмеричной системе) будет представлено следующим образом:

223 : 400000000 170 : 200000000

Бит знака каждого слова равен нулю. Первый порядок 223 (в восьмеричной системе) представляет десятичный порядок 19. Второй порядок 170 представляет десятичный порядок — 8. Соответствующее десятичное число равно

$$.5*2^{19}+.25*2^{-8}=2^{18}+2^{-10}$$

Таким образом, для представления чисел с двойной точностью в KA-процессоре требуется 9 лишних битов. В противоположность этому KI- и KL-процессоры для увеличения точности дробной части используют все биты второго слова, кроме знакового. В этих процессорах 262144.0009765625 запоминается как

223 : 400000000 100000000000

Арифметика вещественных чисел на DEC-10 дает точность 8 десятичных цифр (27 битов). KA-процессоры обеспечивают точность 16 десятичных цифр (54 бита) при работе с числами двойной точности. KI- и KL-процессоры улучшают точность до 18 десятичных цифр (62 бита). Диапазон представления чисел во всех рассмотренных случаях составляет примерно от 10^{-38} до 10^{38} (от 2^{-129} до 2^{127}).

6.10. Организация памяти ЭВМ DEC-10

В отличие от машины IBM 360-370, которые считаются «байт-ориентированными», DEC-10 — «слово-ориентированные» машины, т. е. у них наименьшая адресуемая ячейка памяти — это слово длиной 36 двоичных разрядов. Вещественные, целые и логические переменные могут быть размещены в любом слове памяти. И что особенно ценно, переменным с двойной точностью могут быть отведены любые пары последовательных слов независимо от адреса. Операций над целыми полусловами в DEC-10 нет, логическим переменным нельзя отвести поле памяти длиной менее слова.

Такая ориентация на слово имеет свои достоинства и недостатки. Положительная сторона — программисту не нужно заботиться о выравнивании данных на соответствующие границы, и поэтому упорядочения переменных в инструкциях COMMON и EQUIVALENCE не требуется.

Недостатком такой ориентации является отсутствие гибкости. Целые переменные должны занимать 36 битов даже в том случае, когда их диапазон ограничен небольшими величинами. Что еще хуже, в стандартных компиляторах Фортрана для этих машин не обеспечена возможность работы ни с литерными переменными, ни с эквивалентом переменных типа LOGICAL*1. Для обхода этих последних ограничений при программировании на Фортране для DEC-10 обычно работают с данными,

представленными как числа с фиксированной точкой, и операциями бит-маскирования для упаковки в слово и распаковки из слова пяти литер в коде ASCII (по 7 бит на литеру).

Упражнения

6.01. Запишите 10-разрядное двоичное представление числа —21 в прямом, обратном и дополнительном кодах.

6.02. Запишите все 10-разрядные двоичные представления числа ноль в прямом, обратном и дополнительном кодах.

6.03. Какие десятичные числа представлены 10-разрядным двоичным числом 1111100111 в прямом, обратном и дополнительном кодах?

6.04. Какой результат получится при сложении 10-разрядных двоичных чисел 1111110110 и 0000001101 в прямом, обратном и дополнительном кодах?

6.05. Какие десятичные числа представлены восемью шестнадцатеричными цифрами (если код дополнительный)?

000001FE
FFFFFFEO

6.06. Какие десятичные числа представлены 12 восьмеричными цифрами (если код обратный)?

777777777000
000000001162

6.07. Следующие шестнадцатеричные цепочки представляют два вещественных числа в IBM 360—370:

BFC00000 41FA8000

а) Какие десятичные числа они представляют?
б) Чему равно шестинадцатеричное представление их суммы?

6.08. Следующие восьмеричные цепочки представляют два вещественных числа в DEC-10:

574150000000 207765600000

а) Какие десятичные числа они представляют?
б) Чему равно восьмеричное представление их суммы?

6.09. Следующая шестнадцатеричная цепочка представляет число с двойной точностью в IBM 360—370:

46100000 C0000000

Какое двоичное число представляет эта цепочка?

6.10. Следующая восьмеричная цепочка представляет число с двойной точностью в DEC-10 с процессором KL:

217707000213 330000000000

Какое двоичное число она представляет?

6.11. Предположим, что мы работаем с IBM 360 или 370 и нам нужна общая область, которая содержит каждую из переменных, декларируемых ниже:

```
DOUBLE PRECISION D
INTEGER I
INTEGER*2 K1(3), K2(4)
LOGICAL*1 L1,L2
```

Чтобы правильно выравнять переменные на соответствующие границы, мы описываем общую область так:

```
COMMON D,I, K1(3), K2(4), L1, L2
```

Конечно, существуют и другие декларации, которые обеспечивают правильное выравнивание границ. Приведите все допустимые в данном случае декларации.

Глава 7

СТРУКТУРА ПРОСТОЙ ЭВМ

В этой главе мы кратко рассмотрим основные устройства ЭВМ и затем дадим описание простой гипотетической машины. Наша цель состоит в том, чтобы показать, как выполняются программы, и заложить основы для изучения в гл. 8 компиляторов и загрузчиков.

7.1. Устройства ЭВМ

Большинство ЭВМ состоит из следующих основных четырех устройств: *центрального процессора* (ЦП), *арифметического и*

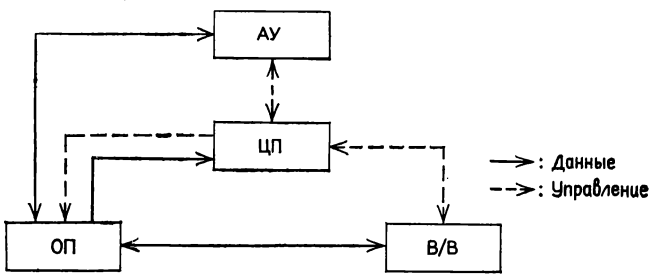


Рис. 7.1. Взаимодействие между устройствами ЭВМ.

логического устройства (АУ), *главной (оперативной) памяти* (ОП) и набора *устройств ввода/вывода* (В/В). Рис. 7.1 иллюстрирует основные взаимосвязи между этими устройствами.

7.1.1. Оперативная память

Это устройство состоит из ячеек, в каждой из которых может храниться число. Числа могут представлять значения данных или команды машинного языка. При этом не проводится никакого различия между ячейками, содержащими данные или команды.

Каждая ячейка оперативной памяти ЭВМ имеет свой адрес. Адреса используются для обращения к данным и передачи управления между командами.

Оперативная память сама по себе не обрабатывает данные. В ней выполняются только операции выборки и записи. Для выполнения этих операций оперативная память связана с устройством управления и с двумя управляющими словами, называемыми *регистром адреса* (РА) и *регистром данных* (РД [1]) (рис. 7.2).

Устройство управления памятью получает запросы от ЦП, АУ или В/В на выполнение операций «выбрать» и «записать». Оно может обрабатывать только один запрос в каждый момент времени. Если запрос поступил в тот момент, когда память недоступна, устройство управления ставит этот запрос в очередь, и он ожидает выполнения.

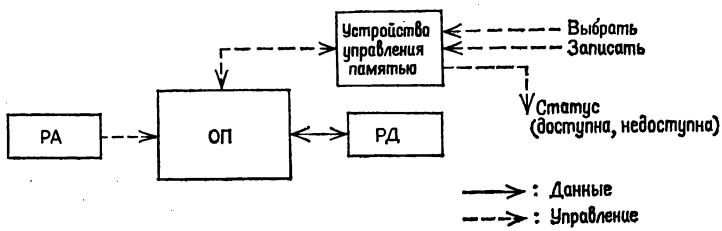


Рис. 7.2. Структура оперативной памяти.

При запросе на запись требуется передать в устройство памяти адрес и элемент данных, который должен быть записан в ячейку памяти с этим адресом. Операция записи выполняется засылкой адреса в регистр РА, данных в регистр РД и последующим указанием устройству памяти произвести запись. При запросе на выборку требуется передать в устройство памяти адрес ячейки, содержимое которой должно быть считано. Операция выборки выполняется передачей адреса в регистр РА и указанием устройству памяти выполнить чтение. Устройство памяти пересылает содержимое ячейки с этим адресом в регистр РД. Наконец, устройство, затребовавшее элемент данных, считывает содержимое регистра РД. Эти действия могут быть расписаны следующим образом:

Запись: Ждать пока станет доступной ОП.
РА ← адрес ячейки для записи данных.
РД ← элемент данных, который требуется записать.
Послать сигнал запроса на запись.

Выборка: Ждать пока станет доступной ОП.
РА ← адрес ячейки, содержащей элемент данных.
Послать сигнал запроса на выборку.
Прочесть содержимое РД.

[1]) В оригинале MAR и MDR; последний иногда называют буферным регистром (MBR).— *Прим. ред.*

7.1.2. Центральный процессор

Основная функция центрального процессора — обеспечить последовательность выполнения команд в программе и управление выполнением каждой отдельной команды программы. Для того чтобы обеспечить последовательность выполнения команд, ЦП взаимодействует с управляющим словом, называемым *программным счетчиком* (ПС). В этот счетчик первоначально заносится адрес первой команды программы и после выполнения каждой команды он изменяется, т. е. туда заносится адрес следующей выполняемой команды.

Приведенная ниже схема показывает упрощенный алгоритм работы ЦП при последовательном выполнении команд программы:

Инициализация:	ПС ← адрес первой команды.
Выборка:	Ждать, пока не станет доступной ОП. РА ← ПС. Послать сигнал запроса на выборку. КОМАНДА ← РД.
Приращение:	ПС ← ПС+1.
Декодирование:	Определить, какого типа команда; если команда недопустимая, прекратить выполнение.
Выполнение:	Управлять выполнением команды.
Цикл:	Передать управление на шаг «Выборка».

Увеличение программного счетчика ПС на 1 означает, что далее будет выполняться следующая по порядку команда. Если эта команда окажется командой передачи управления, то на шаге «Выполнение» произойдет изменение программного счетчика. Действительно, если команда вызывает передачу управления в ячейку АДР, то на шаге «Выполнение» значение АДР будет помещено в регистр ПС. Таким образом может быть изменена последовательность выполнения команд.

7.1.3. Арифметическое и логическое устройство

АУ это та часть ЭВМ, которая выполняет арифметические и логические операции. Устройство обычно всегда содержит аппаратные средства для выполнения арифметических операций (сложения, вычитания, умножения и деления) над целыми числами и операций сравнения целых чисел. Дополнительно многие ЭВМ могут выполнять соответствующие операции над числами с плавающей точкой.

Для реализации вычислений и запоминания полученных результатов АУ имеет несколько ячеек памяти, называемых *регистрами*. Они используются для запоминания операндов и результатов выполнения операций.

7.2. Простейшая машина

7.2.1. Подробное описание машины SADSAC

Для более подробного описания устройств ЭВМ рассмотрим условную десятичную машину SADSAC (SADSAC — сокращение от **S**implified version of **A** **D**ecimal **S**ingle **A**ddress **C**omputer «Упрощенная версия десятичной одноадресной ЭВМ»). Машина имеет следующие характеристики:

Оперативная память. 100 ячеек с адресами от 00 до 99. В каждой ячейке могут храниться три десятичные цифры плюс знак.

Устройства ввода/вывода. Одно устройство ввода с перфокарт и одно печатающее устройство.

Арифметическое устройство. Содержит один 3-разрядный десятичный регистр (со знаком), называемый сумматором (сокращенно СМ).

Формат команд. Каждая команда имеет форму $\pm caa$, где aa — адрес памяти и $\pm c$ — код выполняемой операции.

SADSAC выполняет следующие команды:

± 0: недопустимый код операции;
$+1$: очистить сумматор (СМ);
$+2$: загрузить СМ содержимым ячейки памяти aa;
$+3$: записать содержимое СМ в ячейку памяти aa;
$+4$: прибавить содержимое aa к СМ;
$+5$: вычесть содержимое aa из СМ;
$+6$: умножить СМ на содержимое aa;
$+7$: разделить СМ на содержимое aa;
$+8$: прочитать целую величину со следующей перфокарты и записать ее в aa;
$+9$: напечатать содержимое aa;
-1: передать управление в ячейку aa, т. е. занести aa в программный счетчик;
-2: передать управление в aa, если СМ$=0$;
-3: передать управление в aa, если СМ>0;
-4: передать управление в aa, если СМ<0;
-5: передать управление в aa, если в результате выполнения последней арифметической операции получилось число,

превышающее максимально допустимое (эта ситуация называется *переполнением*);

—6: передать управление в *aa*, если СМ≠0;

—7: передать управление в *aa*, если СМ≤0;

—8: передать управление в *aa*, если СМ≥0;

—9: остановить выполнение программы.

Приведенный набор команд, конечно, проще, чем в любой реальной ЭВМ, однако он достаточен для логического анализа фортранных программ. Основными ограничениями здесь являются длина слова, размер памяти и трудности, возникающие при выполнении действий над числами с плавающей точкой. Проиллюстрируем работу переведенной на машинный язык SADSAC следующей простой фортранной программы:

```
C**** ВЫЧИСЛЕНИЕ ФАКТОРИАЛА N ****
      INTEGER N, FACT
      READ (5,10) N
   10 FORMAT ( I12 )
      FACT = 1
      IF (N .LE. 1) GO TO 30
      DO 20 I = 2,N
         FACT = FACT * I
   20 CONTINUE
   30 WRITE (6,10) FACT
      STOP
      END
```

Ячейки памяти	Содержимое	Примечания
00	+814	Прочитать значение N.
01	+217	Если значение N меньше или равно
02	+514	единице, перейти к печати результата 1.
03	—812	
04	+216	
05	+615	Умножить следующий множитель на те-
06	+315	кущее значение и сохранить его.
07	+216	FACT.
08	+417	Вычислить следующий множитель и сох-
09	+316	ранить его в 1.
10	+514	Если вычисление не закончено, то пов-
11	—704	торить умножение.
12	+915	Напечатать значение факториала N.
13	—900	Останов.
14	+000	N.
15	+001	FACT (начальное значение 1).
16	+002	Индекс цикла 1, первоначально 2.
17	+001	Константа 1.
18	несущественно	
.		
.		
.		
.		
99		

Программирование на машинном языке утомительно и сопряжено с ошибками. В программе трудно делать изменения, так как из-за одной незначительной модификации может появиться большое число неверных адресов. Например, если бы мы решили напечатать значение N, как только оно прочитано, следовало бы вставить команду +914 в ячейку 01. Эта вставка привела бы к смещению вниз на одну ячейку всех последующих команд и данных. Такое смещение потребовало бы изменить адрес переходов и адреса данных. В частности, адрес ячейки N был бы не 14, а 15. Для того чтобы программы было легче писать и изменять, программисты обычно составляют машинно-ориентированные программы не на машинном языке, а на *языке ассемблера*.

7.2.2. Язык ассемблера для машины SADSAC

В языке ассемблера можно пользоваться символической адресацией памяти, а также мнемоническими обозначениями для кодов операций. Допустим, мы разработали для машины SADSAC язык, в котором команды имеют следующий формат:

Колонка	1—8	10—13	15—22	25—72
Назначение	*метка*	*КОП*	*операнд*	*комментарий*

где *метка* содержит от одной до восьми летер; *КОП* — код операции, содержит от одной до четырех букв и является мнемоническим обозначением машинной команды или команд CON, END; *операнд* зависит от кода операции; *комментарий* — любой текст.

Дозволенные коды операций языка ассемблера и их значение приведены ниже:

Мнемоническое обозначение	Эквивалентный код машинной команды	Операнд
ZERO	+1	*отсутствует*
LOAD	+2	метка
STOR	+3	метка
ADD	+4	метка
SUB	+5	метка
MUL	+6	метка
DIV	+7	метка
READ	+8	метка
PRNT	+9	метка
BR	—1	метка
BRZ	—2	метка
BRP	—3	метка
BRM	—4	метка
BRO	—5	метка
BRNZ	—6	метка
BRNP	—7	метка
BRNM	—8	метка
STOP	—9	*отсутствует*

CON	определяет начальное значение данных	целая величина
END	отмечает физический конец программы	*отсутствует*

Программа на языке ассемблера, эквивалентная программе вычисления факториала N, написанной на машинном языке, приведена ниже:

```
      READ N       ВВЕСТИ ЗНАЧЕНИЕ N.
      LOAD ONE     ЕСЛИ N МЕНЬШЕ ИЛИ РАВНО 1,
      SUB N        ТО ПЕРЕЙТИ К DONE.
      BRNM DONE
NEXT  LOAD I       УМНОЖИТЬ ЗНАЧЕНИЕ FACT НА
      MUL FACT     СЛЕДУЮЩИЙ ЧЛЕН ПОСЛЕДОВАТЕЛЬНОСТИ.
      STOR FACT
      LOAD I
      ADD ONE      ВЫЧИСЛИТЬ НОВЫЙ ЧЛЕН.
      STOR I
      SUB N        ЕСЛИ ОН ПРЕВЫШАЕТ N, ПЕРЕЙТИ К
      BRNP NEXT    DONE ЕСЛИ НЕТ, ПРОДОЛЖИТЬ
DONE  PRNT FACT    ВЫЧИСЛЕНИЯ.DONE — НАПЕЧАТАТЬ РЕЗУЛЬ-
      STOP         ТАТ.
N     CON 0        ЗДЕСЬ ХРАНИТСЯ N.
FACT  CON 1        ЗДЕСЬ ХРАНИТСЯ ФАКТОРИАЛ.
I     CON 2        ПЕРВЫЙ ЧЛЕН ФАКТОРИАЛА.
ONE   CON 1        КОНСТАНТА 1.
      END
```

Программа стала намного понятней. Заметим, насколько легче модифицировать программу на языке ассемблера. Например, если бы потребовалось напечатать значение N после его считывания, мы бы просто вставили команду PRNT N после команды READ. Никакие другие команды в связи с внесением этого добавления не нужно изменять.

При изучении SADSAC невольно возникает вопрос: как выполнять операции над массивами? Так как индексами воспользоваться нельзя, то может показаться, что мы ограничиваем вычисления только простыми переменными. Это не так. Принцип обработки массивов основан на том, что ячейки памяти, содержащие команды, ничем не отличаются от ячеек, в которых содержатся данные. Таким образом, команды можно обрабатывать как данные, т. е. их можно изменять. Рис. 7.3 иллюстрирует эту возможность. На рисунке изображены две эквивалентные программы: одна на Фортране, другая на языке ассемблера машины SADSAC. Программа считывает данные и печатает их в обратном порядке.

```
C*** ПЕЧАТЬ ВХОДН. ДАННЫХ В ОБРАТН. ПОРЯДКЕ
     INTEGER LIST(10), N
     READ (5,10) N
  10 FORMAT ( I12 )
     DO 20 I = 1, N
        READ (5,10) LIST(I)
  20 CONTINUE
     DO 30 I = 1, N
        K = N+1 - I
        WRITE (6,10) LIST(K)
  30 CONTINUE
     STOP
     END
```

(а) Программа на Фортране, печатающая входные данные в обратном порядке

```
         READ   N        КОЛИЧЕСТВО ДОПОЛНИТЕЛЬНО СЧИТЫВАЕ-
                         МЫХ КАРТ
NEXTRD   LOAD   I        ИЗМЕНЕНИЕ READ ДЛЯ ОБРАЩЕНИЯ К
         ADD    RD1      СЛЕДУЮЩЕМУ ЭЛЕМЕНТУ МАССИВА
         STOR   RD2
RD2      READ   LIST     ** ЭТА ИНСТРУКЦИЯ READ
                         МОДИФИЦИРУЕТСЯ **
         LOAD   I        УВЕЛИЧИТЬ СЧЕТЧИК КАРТ И ПОСМОТРЕТЬ,
         ADD    ONE      ЕСТЬ ЛИ ЕЩЕ КАРТЫ
         STOR   I
         SUB    N        ЕСЛИ ЕСТЬ, ПЕРЕДАТЬ УПРАВЛЕНИЕ НА
         BRM    NEXTRD   ВВОД СЛЕДУЮЩЕЙ ПЕРФОКАРТЫ
NEXTWR   LOAD   I        ВВЕДЕНЫ ВСЕ КАРТЫ, ВСЕ ЛИ НАПЕЧАТАНО?
         SUB    ONE
         BRM    DONE
         STOR   I        ЕСЛИ ДА, ИЗМЕНИТЬ PRINT ДЛЯ ОБРАЩЕ-
         ADD    WR1      НИЯ К СЛЕДУЮЩЕМУ ЭЛЕМЕНТУ, ДВИГАЯСЬ
                         НАЗАД.
         STOR   WR2
WR2      PRNT   LIST     ** ЭТА ИНСТРУКЦИЯ МОДИФИЦИРУЕТСЯ **
         BR     NEXTWR   ВЗЯТЬ СЛЕДУЮЩЕЕ ЗНАЧЕНИЕ
DONE     STOP            ВЫХОД ПО ОКОНЧАНИИ РАБОТЫ
RD1      READ   LIST     ИСПОЛЬЗУЕТСЯ ДЛЯ ФОРМИРОВАНИЯ READ
WR1      PRNT   LIST     ИСПОЛЬЗУЕТСЯ ДЛЯ ФОРМИРОВАНИЯ PRINT
N        CON    0        ЧИСЛО СЧИТЫВАЕМЫХ КАРТ
I        CON    0        СЧЕТЧИК ДЛЯ ЦИКЛОВ
ONE      CON    1        ПОЛЕЗНАЯ КОНСТАНТА
LIST     CON    0        МАССИВ ДЛЯ ЗНАЧЕНИЙ ПЕРФОКАРТ
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         CON    0
         END
```

(б) Эквивалентная SADSAC-программа

Рис. 7.3. Обработка массивов в SADSAC.

7.2.3. Моделирование машины SADSAC

Одна из интересных особенностей цифровых ЭВМ состоит в том, что на них можно моделировать операции других ЭВМ. Мы продемонстрируем эту возможность с помощью программы на Фортране (рис. 7.4), которая использует в качестве входной ин-

формации программу на машинном языке SADSAC и ее данные. На выходе распечатывается память машины SADSAC, какой она была до выполнения программы, строки, которые должны были быть выведены на печать в ходе работы SADSAC-программы, и, наконец, дампинг памяти после выполнения программы. Останов SADSAC-программы может произойти по одной из следующих причин:

1. Выполнена команда STOP.
2. Обнаружен неверный код операции (+0 или —0).
3. Была попытка разделить на ноль.
4. Нарушены границы памяти (обращение к несуществующей ячейке, например 100)
5. Была сделана попытка прочитать данные после окончания ввода (уже была считана запись «конец файла»).

Вводимая в имитатор SADSAC-программа (на картах) должна быть представлена в следующем формате:

Колонки 2, 3. Целая величина в диапазоне между 0 и 99, представляющая адрес, связанный с этой перфокартой.

Колонки 6—9. Целая величина со знаком или без знака в диапазоне между —999 и +999, представляющая значение, которое должно быть записано в память. Если эта величина выходит за допустимые размеры, выдается сообщение об ошибке и выполнение задания прекращается.

Конец SADSAC-программы отмечается на перфокарте * в колонке 1. Все другие колонки перфокарты игнорируются. Любые данные, считываемые SADSAC-программой, располагаются за перфокартой с ограничителем *.

```
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C *  ПРОГРАММА МОДЕЛИРОВАНИЯ SADSAC                                *
C *    ЭТА ПРОГРАММА ВВОДИТ НАБОР ПЕРФОКАРТ, ОПРЕДЕЛЯЮЩИХ          *
C *    НАЧАЛЬНОЕ СОДЕРЖИМОЕ ПАМЯТИ МАШИНЫ. КАЖДАЯ ВХОДНАЯ          *
C *    КАРТА ПРЕДСТАВЛЕНА В ФОРМЕ:                                  *
C *      КОЛОНКИ 2—3: НОМЕР ЯЧЕЙКИ, ЗАНИМАЕМОЙ ДАННЫМ              *
C *      СЛОВОМ.                                                    *
C *      КОЛОНКИ 6—9: СОДЕРЖИМОЕ ЭТОГО СЛОВА.                       *
C *                                                                 *
```

Рис. 7.4. Программа моделирования SADSAC.
(Текст в инстр. **15**: «** Ошибка ** Неправильная команда — », «в ячейке»; **31**: «**Ошибка ** Неожиданный конец файла»; **45**: «**Программа не выполняется в виду ошибки **»; **110**: «Содержимое ячейки»; **445**: «**Нормальное завершение — выполнен STOP **»; **556**: «**Ошибка ** Неправильный код операции»; **667**: «**Ошибка ** Значение PC больше 99»; **778**: «** Ошибка ** Деление на ноль»; **889**: «** Ошибка ** Неожиданный конец файла»; **10**: «** Дамп системы SADSAC ** », «Программный счетчик», «Сумматор», «Содержимое памяти»; **15**: «Ячейки».)

```
C *  КОНЕЦ ВСЕХ ТАКИХ ПЕРФОКАРТ ОТМЕЧАЕТСЯ КАРТОЙ, СОДЕРЖА-  *
C *    ЩЕЙ * В КОЛОНКЕ 1. ЭТА ПЕРФОКАРТА НЕ СОДЕРЖИТ НИКАКИХ  *
C *    ДАННЫХ. ЗА КАРТОЙ СО * СЛЕДУЕТ НАБОР КАРТ, СЧИТЫВАЕ-   *
C *    МЫХ ПОЛЬЗОВАТЕЛЬСКОЙ ПРОГРАММОЙ НА ЯЗЫКЕ МАШИНЫ        *
C *    SADSAC.                                                 *
C *                                                            *
C *  СОЗДАВАЕМЫЙ ВЫХОД:                                        *
C *      ДАМП ПАМЯТИ ПОСЛЕ НАЧАЛЬНОЙ ЗАГРУЗКИ ПРОГРАММЫ;      *
C *      ТРАССИРОВКА ВЫПОЛНЕНИЯ ВСЕХ КОМАНД (ЕСЛИ TRACE=      *
C *      .TRUE.);                                              *
C *      ВЫВОД ПО КАЖДОЙ КОМАНДЕ PRNT В ПРОГРАММЕ ПОЛЬЗО-      *
C *      ВАТЕЛЯ;                                               *
C *      ДАМП КОНЕЧНОГО СОДЕРЖИМОГО ПАМЯТИ;                    *
C       СООБЩЕНИЯ ОБ ОШИБКАХ.                                  *
C *                                                            *
C *  ОБЩИЕ ПЕРЕМЕННЫЕ:                                         *
C *      MEM    — СПИСОК ТЕКУЩЕГО СОДЕРЖИМОГО ПАМЯТИ;          *
C *      PC     — ПРОГРАММНЫЙ СЧЕТЧИК;                         *
C *      AC     — СУММАТОР;                                    *
C *      PRT    — НОМЕР УСТРОЙСТВА ПЕЧАТИ;                     *
C *      RDR    — НОМЕР УСТРОЙСТВА ВВОДА;                      *
C *      TRC    — НОМЕР УСТРОЙСТВА ДЛЯ ВЫВОДА ТРАССИРОВКИ;     *
C *      OVFLOW— ЕСЛИ TRUE, ТО ПОСЛЕДНЯЯ АРИФМЕТИЧЕСКАЯ       *
C *               ОПЕРАЦИЯ ВЫЗВАЛА ПЕРЕПОЛНЕНИЕ;               *
C *      TRACE  — УСТАНАВЛИВАЕТСЯ В TRUE, ЕСЛИ НЕОБХОДИМА      *
C *               ТРАССИРОВКА.                                 *
C *                                                            *
C *  ВЫЗЫВАЕМЫЕ ПОДПРОГРАММЫ:                                  *
C *      DUMP   — РАСПЕЧАТЫВАЕТ СОДЕРЖИМОЕ ПАМЯТИ;             *
C *      INTERP — ИНТЕРПРЕТИРУЕТ SADSAC-ПРОГРАММУ.             *
C *                                                            *
C *  ПРОГРАММИСТ: — — —              ДАТА: — — —               *
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


      INTEGER LOC, STMT, DELIM, DEL
      LOGICAL EXEC
      LOGICAL OVFLOW, TRACE
      INTEGER MEM(100), PC, AC, PRT, RDR, TRC
      COMMON/SADSAC/ MEM, PC, AC, PRT, RDR, TRC, OVFLOW, TRACE
      DATA    DEL/'*'/, EXEC/.TRUE./
C


C       ВВЕСТИ СЛЕДУЮЩУЮ КАРТУ SADSAC-ПРОГРАММЫ
10    READ (RDR,11,END=30) DELIM, LOC, STMT
11    FORMAT ( A1, I2, 2X, I4 )
C
C       КОНЕЦ ПРОГРАММЫ ПОЛЬЗОВАТЕЛЯ?
      IF ( DELIM .EQ. DEL ) GO TO 40
C
C       ПРАВИЛЬНО ЛИ ЗАПИСАНА МАШИННАЯ КОМАНДА?
      IF ( IABS(STMT) .LT. 1000 ) GO TO 20
C
C       ОШИБКА, ПРОГРАММА НЕ МОЖЕТ БЫТЬ ВЫПОЛНЕНА
      EXEC = .FALSE.
      WRITE (PRT,15) STMT, LOC
15    FORMAT ('0** ERROR ** INVALID COMMAND - ', I4, 'AT LOCATION', I2)
      GO TO 10
C
C       КОМАНДА ВЕРНАЯ - ЗАПИСАТЬ ЕЕ
20    MEM(LOC+1) = STMT
C
C       ВЕРНУТЬСЯ К ЧТЕНИЮ СЛЕДУЮЩЕЙ КАРТЫ
      GO TO 10
```

Продолжение Рис. 7.4.

```
C
C       ОШИБКА, НЕОЖИДАННЫЙ КОНЕЦ ФАЙЛА
30    WRITE (PRT,31)
31    FORMAT ('0** ERROR ** UNEXPECTED END-OF-FILE')
      STOP
C
C       КОНЕЦ ПРОГРАММЫ - ЕСЛИ В НЕЙ ОШИБКА, ОСТАНОВ
40    IF ( EXEC ) GO TO 50
      WRITE (PRT,45)
45    FORMAT ('0**
      STOP
C
C       НАЧИНАЕТСЯ ВЫПОЛНЕНИЕ - ДАМП ПАМЯТИ, ИНТЕРПРЕТАЦИЯ
50    CALL DUMP
      CALL INTERP
C
C       ЗАВЕРШАЮЩИЙ ДАМП
      CALL DUMP
      STOP
      END


      SUBROUTINE INTERP
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИНТЕРПРЕТИРОВАТЬ ПРОГРАММУ, ПОКА НЕ ВСТРЕТИТСЯ STOP ИЛИ ОШИБКА.     *
C* ТРАССИРОВКА КОМАНД, ЕСЛИ 'TRACE' - TRUE                              *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER ADDR, INSTR, OPCODE, MM, OPTYP, VALUE
      LOGICAL OVFLOW, TRACE
      INTEGER MEM(100), PC, AC, PRT, RDR, TRC
      COMMON/SADSAC/ MEM, PC, AC, PRT, RDR, TRC, OVFLOW, TRACE
C
C*********************************************************************
C****            ВЫПОЛНИТЬ СЛЕДУЮЩУЮ КОМАНДУ                        ****
C       ЕСЛИ МЫ ВЫШЛИ ЗА ПРЕДЕЛЫ ПАМЯТИ, ОСТАНОВ
40    IF ( PC .GT. 99 ) GO TO 666
C
C       ДЕКОДИРОВАТЬ СЛЕДУЮЩУЮ КОМАНДУ
80    INSTR = MEM(PC+1)
      OPCODE = INSTR/100
      ADDR = IABS(INSTR - OPCODE*100)
      MM = MEM(ADDR+1)
      IF ( TRACE ) WRITE (TRC,90) PC, INSTR, MM, AC
90    FORMAT (' ** TRACE ** PC=', I2, ':IN=', I4, ':MM=', I4, ':AC=',I4)
      PC = PC + 1
C
C       ПЕРЕХОД НА ИНТЕРПРЕТАЦИЮ ДАННОЙ КОМАНДЫ
      OPTYP = OPCODE + 10
      GO TO (444,208,207,206,205,204,203,202,201,555,
     X       101,102,103,104,105,106,107,108,109),OPTYP
C
```

Продолжение Рис. 7.4.

```
C
C               OPCODE = 1  --  ОЧИСТИТЬ СУММАТОР
101   AC = 0
      GO TO 40
C               OPCODE = 2  --  LOAD
102   AC = MM
      GO TO 40
C               OPCODE = 3  --  STOR
103   MEM(ADDR+1) = AC
      GO TO 40
C               OPCODE = 4  --  ADD
104   AC = AC + MM
      GO TO 150
C               OPCODE = 5  --  SUB
105   AC = AC - MM
      GO TO 150
C               OPCODE = 6  --  MUL
106   AC = AC * MM
      GO TO 150
C               OPCODE = 7  --  DIV
C               ПРОВЕРИТЬ, НЕ ПРОИСХОДИТ ЛИ ДЕЛЕНИЕ НА НОЛЬ
107   IF ( MM .EQ. 0 ) GO TO 777
      AC = AC / MM
      GO TO 40
C               OPCODE = 8  --  READ
C               ЗДЕСЬ БЕСФОРМАТНЫЙ ВВОД WATFIV. ЕСЛИ ВАШ
C               КОМПИЛЯТОР НЕ ДОПУСКАЕТ ЕГО, ТО МОЖНО
C               ВОСПОЛЬЗОВАТЬСЯ ПОДПРОГРАММОЙ EDCHIN, СМ. ГЛ.3
108   READ (RDR,*,END=888) VALUE
      MEM(ADDR+1) = MOD(VALUE,1000)
      GO TO 40
C               OPCODE = 9  --  PRNT
109   WRITE (PRT,110) ADDR, MM
110   FORMAT (' CONTENTS OF LOCATION ', I2, ' ARE ', I4)
      GO TO 40


C
C               ЕСЛИ ПЕРЕПОЛНЕНИЕ, УСТАНОВИТЬ ФЛАГ ПЕРЕПОЛНЕНИЯ
150   OVFLOW = .FALSE.
      IF (IABS(AC) .LE. 999) GO TO 40
      AC = MOD(AC,1000)
      OVFLOW = .TRUE.
      GO TO 40
C
C               OPCODE = -1  --  BR
201   PC = ADDR
      GO TO 80
C               OPCODE = -2  --  BRZ
202   IF ( AC .EQ. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -3  --  BRP
203   IF ( AC .GT. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -4  --  BRM
204   IF ( AC .LT. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -5  --  BRO
205   IF ( OVFLOW ) PC = ADDR
      GO TO 80
C               OPCODE = -6  --  BRNZ
206   IF ( AC .NE. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -7  --  BRNP
207   IF ( AC .LE. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -8  --  BRNM
208   IF ( AC .GE. 0 ) PC = ADDR
      GO TO 80
C               OPCODE = -9  --  STOP
444   WRITE (PRT,445)
445   FORMAT ('0**
      GO TO 999
```

Продолжение Рис. 7.4.

```
C
C             ОКОНЧАНИЯ ПО ОШИБКЕ
C
C             ПОПЫТКА ВЫПОЛНИТЬ OPCODE = 0
555   WRITE (PRT,556)
556   FORMAT ('0** ERROR ** INVALID OPERATION CODE')
      GO TO 999
C             PC ПОЛУЧИЛА ЗНАЧЕНИЕ БОЛЬШЕ 99
666   WRITE (PRT,667)
667   FORMAT ('0** ERROR ** PC VALUE GREATER THAN 99')
      GO TO 999
C             ПОПЫТКА ДЕЛЕНИЯ НА НОЛЬ
777   WRITE (PRT,778)
778   FORMAT ('0** ERROR ** DIVIDE BY ZERO')
      GO TO 999
C             НЕОЖИДАННЫЙ КОНЕЦ ФАЙЛА
888   WRITE (PRT,889)
889   FORMAT ('0** ERROR ** UNEXPECTED END-OF-FILE')
999   RETURN
      END



      SUBROUTINE DUMP
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* РАСПЕЧАТАТЬ СОДЕРЖИМОЕ ПАМЯТИ SADSAC И ЗНАЧЕНИЯ PC И AC          *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INTEGER LOC1, LOC2
      LOGICAL OVFLOW, TRACE
      INTEGER MEM(100), PC, AC, PRT, RDR, TRC
      COMMON/SADSAC/ MEM, PC, AC, PRT, RDR, TRC, OVFLOW, TRACE
C
C             ДАМП СОСТОЯНИЯ СИСТЕМЫ
      WRITE (PRT,10) PC, AC
10    FORMAT (///, T40, '** SADSAC SYSTEM DUMP **' //
     1        T16, 'PROGRAM COUNTER=', I3, T68, 'ACCUMULATOR=', I4 //
     2        T45, 'MEMORY CONTENTS' //)
C
C             ДАМП ПАМЯТИ
      DO 20 I = 1, 91, 10
         J = I + 9
         LOC1 = I - 1
         LOC2 = LOC1 + 9
         WRITE (PRT,15) LOC1, LOC2, (MEM(K), K=I, J)
15       FORMAT (T13, 'LOCATIONS ', I2, '-', I2, 10I6)
20    CONTINUE
      RETURN
      END

      BLOCK DATA
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* ИНИЦИАЛИЗАЦИЯ ОБЩИХ ОБЛАСТЕЙ:                                     *
C*       MEM     = 0    :ВСЕ ЯЧЕЙКИ ПАМЯТИ СНАЧАЛА НУЛЕВЫЕ          *
C*       PC      = 0    :ЗАПУСК ПРОГРАММЫ С ПЕРВОЙ ЯЧЕЙКИ            *
C*       AC      = 0    :СНАЧАЛА В СУММАТОРЕ НОЛЬ                    *
C*       PRT     = 6    :НОМЕР УСТРОЙСТВА PRINTER IBM 360-370        *
C*       RDR     = 5    :НОМЕР УСТРОЙСТВА READER IBM 360-370         *
C*       TRC     = 6    :НОМЕР УСТРОЙСТВА PRINTER IBM 360-370        *
C*       OVFLOW =FALSE:ВНАЧАЛЕ НЕТ ПЕРЕПОЛНЕНИЯ                      *
C*       TRACE  = TRUE:ВКЛЮЧИТЬ ОТЛАДОЧНУЮ ТРАССИРОВКУ               *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      LOGICAL OVFLOW, TRACE
      INTEGER MEM(100), PC, AC, PRT, RDR, TRC
      COMMON/SADSAC/ MEM, PC, AC, PRT, RDR, TRC, OVFLOW, TRACE
      DATA MEM/100*0/, PC/0/, AC/0/
      DATA PRT/6/, RDR/5/, TRC/6/, TRACE/.TRUE./, OVFLOW/.FALSE./
      END
```

Продолжение Рис. 7.4.

Упражнения

7.01. Напишите программу на языке ассемблера SADSAC, которая читает величины A, B и C, выполняет инструкцию Фортрана

A=((B—C)*(A—C))/(B+C)

и результат вычисления выводит на печать.

7.02. Предположим, что мы имеем недорогую модель машины SADSAC. В этой машине выполняется только одна арифметическая операция — вычитание (SUB, машинный код операции +5). Операции ZERO, ADD, MUL и DIV в этой машине не реализованы. Имеется еще операция STOR. Операции LOAD в машине нет, но она может быть заменена последовательностью операций:

```
STOR  TEMP
SUB   TEMP   очистили сумматор
SUB   X      получили —X
STOR  TEMP
SUB   TEMP   теперь получили 0
SUB   TEMP   наконец загрузили X
```

Пользуясь только командами STOR и SUB, напишите программу для вычисления

E=A+2*B—C

Указание: Прежде чем займетесь данным выражением, составьте программы, выполняющие операции сложения и умножения.

7.03. Рассмотренная нами машина SADSAC одноадресная, т. е. в каждой команде можно обращаться только к одному слову памяти. Некоторые машины включают также команды с двумя операндами. В этих командах сумматор явно не указывается. Рассмотрим машину, в которой могут выполняться следующие команды:

ADD A,B — Прибавить A к B и результат запомнить в A.
SUB A,B — Вычесть B из A и результат запомнить в A.
MUL A,B — Умножить A на B и результат запомнить в A.
DIV A,B — Разделить A на B и результат запомнить в A.
MOV A,B — Переслать содержимое B в A.

Пользуясь этими командами, напишите программу для вычисления

X=((Y—Z)*(X—Z))/(Y+Z)

Можно предположить, что X, Y и Z определены. Если вам потребуются ячейки памяти для промежуточных результатов, назовите их T1, T2, ...

7.04. Напишите программы на языке ассемблера SADSAC, моделирующие каждую из 5 команд, приведенных в упр. 7.03.

7.05. Напишите программу на языке ассемблера SADSAC, которая вычисляет наибольший общий делитель двух вводимых чисел A и B.

Глава 8

ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ СРЕДСТВ ОПЕРАЦИОННОЙ СИСТЕМЫ

В предыдущих главах рассматривались методы программирования и дополнительные возможности языка Фортран. Главу 7 мы посвятили структуре ЭВМ и показали, как эта структура может повлиять на фортранные программы. В данной главе мы проанализируем этапы работы ЭВМ между вводом исходной программы и окончанием ее выполнения.

8.1. Операционные системы

Первые ЭВМ, созданные в конце 40-х и в начале 50-х годов, были по современным стандартам медленными, малыми, для них было трудно программировать, неудобно было и прогонять программы. Как результат этими машинами пользовалось очень ограниченное число людей. Типичные ЭВМ того времени похожи на машину SADSAC из гл. 7. Пользователи программировали на машинном языке или на языке ассемблера. Каждый пользователь сам составлял программы, управлял ходом их выполнения и сам собирал результаты выполнения программы. Одновременно мог работать на машине обычно только один человек.

Три важных усовершенствования в середине 50-х годов изменили методы работы с ЭВМ. Первое — создание языков высокого уровня, таких, как Фортран и Кобол, и компиляторов для этих языков. Применение компиляторов облегчило программирование, так что большее число людей стало смотреть на ЭВМ как на полезный инструмент для решения своих задач. Второе — благодаря развитию аппаратных средств быстродействие увеличилось в тысячи раз. В связи с этим стали доступными для решения на ЭВМ более сложные задачи. Третье — были созданы библиотеки программ и подпрограмм, которыми могли пользоваться все, кто работал с машиной. Создание библиотек означало, что для извлечения квадратного корня или вычисления синуса уже не нужно было писать собственные программы. Такие программы мог написать кто-то один, поместить их в общую библиотеку, а все остальные пользователи могли теперь обращаться к этим программам. Эти три важных усовершенствования расширили до-

ступ к ЭВМ, но именно поэтому увеличили потребность в машинном времени.

Из-за возросших потребностей в машинном времени стало расточительством вручную организовывать запуск и следить за выполнением каждого задания. Часто время, затрачиваемое на «ручные» операции, превышало время выполнения программы. Различие в скоростях между человеком и машиной привело к разработке управляющих программ, которые взяли на себя некоторые функции по организации работы на машине. Такие программы были названы *мониторами*. Монитор вводил одну или несколько программ пользователей; вызывал необходимые компиляторы, загрузчики и служебные подпрограммы; передавал управление очередной программе пользователя, получая обратно управление после окончания выполнения программы. Другими словами, монитор обслуживал пользователей. Последовательное выполнение заданий под управлением программы монитора было названо *пакетной обработкой*.

Дальнейшее развитие технологии позволило организовать частичное совмещение операций ввода/вывода с вычислениями. Операции ввода и вывода, особенно ввод с перфокарт и вывод на печать, сопряжены с различными механическими действиями, такими, например, как перемещение перфокарт после чтения. Скорость, с которой перфокарты могут перемещаться, чрезвычайно мала по сравнению со скоростью работы электронных схем ЦП. В связи с тем что выполнение программы, как правило, нельзя продолжить, пока не закончены операции ввода/вывода, ЦП во время этих операций в течение длительного времени простаивает. Такая ситуация потребовала разработки *мультипрограммирования*, при котором одновременно в оперативной памяти находилось несколько программ. Поскольку в машине был только один ЦП, в каждый момент времени только одна программа могла быть активной: другие программы могли, однако, быть готовы занять ЦП в любое время, как только выполнение первой программы приостанавливалось в ожидании завершения операций ввода/вывода. Поэтому две или даже более двух программ одновременно могли находиться в процессе выполнения; каждая работала, пока другая ожидала завершения операции ввода/вывода. Таким путем непроизводительное время ЦП было уменьшено.

С развитием мультипрограммирования управляющие программы-супервизоры превратились из пассивных мониторов в более активные программы. При одновременном обслуживании нескольких пользователей потребовалось управлять распределением ресурсов; супервизор следил за тем, кто владеет ресурсами, и предоставлял имеющиеся в наличии ресурсы запрашивающим их программам. Супервизор, называемый теперь *операционной си-*

стемой, стал ответственным за обеспечение пользователей разделяемыми и неделимыми ресурсами, включая компиляторы, память и устройства В/В. Естественным расширением явилась операционная система, которая планирует выполнение программ пользователей, обеспечивая тем самым наиболее целесообразное использование ресурсов системы. Задание, в котором невелик объем ввода/вывода, но интенсивно используется ЦП, можно сочетать с заданием, в котором выполняется много операций ввода/вывода. Операционная система стала управлять выполнением целой группы программ вместо того, чтобы обслуживать запросы одной программы.

При переходе от режима одиночного пользователя к мультипрограммной пакетной обработке «личные контакты» с ЭВМ были утеряны. Прежде пользователь мог предусмотреть в программе прием определенных входных данных с пульта управления, чтобы с их помощью изменить ход выполнения программы. Так как скорости ЭВМ увеличились и появилось мультипрограммирование, для машины стало слишком дорогим время ожидания ввода данных с пишущей машинки. Растрачивалось также и время программиста, поскольку невозможно было предсказать момент, когда задание начнет выполняться.

Активное взаимодействие человека с ЭВМ в процессе вычислений желательно при решении многих задач. Применение *режима с разделением времени* обеспечило такое взаимодействие без чрезмерных затрат. Разделение времени — это система, при помощи которой многие пользователи одновременно взаимодействуют с машиной, причем у каждого создается впечатление, что исключительно он один на ней работает. Поскольку ЭВМ оперирует намного быстрее человека, она может обслуживать несколько других запросов за время, затрачиваемое пользователем на ввод одного запроса. Для взаимодействия с машиной применяются терминалы, обычно печатающие машинки или дисплеи с клавиатурой. Некоторые ЭВМ обеспечивают одновременно и пакетную обработку, и разделение времени.

8.2. Компиляторы

На современных ЭВМ задачи программируют в основном на языках высокого уровня. В связи с этим были созданы различные компиляторы для удовлетворения потребностей программистов, работающих в различных областях. К компиляторам предъявляются следующие требования:

1. Небольшой размер — для применения на машинах с ограниченной оперативной памятью.

2. Быстрая компиляция — для применения в тех случаях, когда высоки потребности в машинном времени или компилируется большое число программ.
3. Широкие возможности обнаружения ошибок и выдачи понятных диагностических сообщений; особенно полезны в ситуациях, когда многие пользователи (например, студенты, начинающие изучать программирование) не очень хорошо знакомы с исходным языком.
4. Быстрота выполнения или небольшой размер скомпилированной программы — желательны, если скомпилированная программа должна выполняться неоднократно или если для выполнения программы требуется длительный период времени.

Некоторые из перечисленных требований противоречат друг другу, и поэтому конкретный компилятор может удовлетворять одним требованиям за счет других. Компилятор Фортрана уровня H (IBM) может создавать небольшую и быстро выполняемую программу, но сам компилятор громоздкий и медленный. Компиляторы, которые порождают «оптимизированный» код, часто применяют для компиляции так называемых «производственных» программ. Эти программы выполняются многократно. Компилятор WATFIV, с другой стороны, создает относительно неэффективные программы, но быстро их компилирует. Он удобен для студенческой среды, в которой большое количество коротких программ выполняется, как правило, только один раз.

8.3. Объектный код

Компилятор обрабатывает исходную программу и преобразует ее в последовательность машинных команд. Код, создаваемый компилятором, называют *объектным кодом*. Объектный код запоминается на устройствах вспомогательной памяти. Полученный в результате однократной компиляции объектный код можно длительное время хранить и выполнять неограниченное число раз. Если размер программы или время ее выполнения значительны, то применение оптимизированного объектного кода может дать большую экономию машинного времени и памяти.

Подпрограммы Фортрана представляют собой раздельно компилируемые модули и располагаются в памяти отдельно от других подпрограмм. В большинстве вычислительных машин можно объединять объектный код вновь скомпилированной подпрограммы с кодом других, ранее скомпилированных подпрограмм, что позволяет при изменениях в исходной программе повторно компилировать не всю программу, а чаще всего только одну подпрограмму. Это еще один факт в пользу создания модульных программ на основе подпрограмм.

Объектный код программы можно хранить на вспомогательном запоминающем устройстве или выводить на перфокарты. В последнем случае колода перфокарт называется *объектной колодой*. Объектная колода может быть представлена на выполнение иногда совместно с другими программами, написанными на исходном языке. Перфокарты служат недорогим средством для длительного хранения объектных модулей.

8.4. Перемещение программ

Рассмотренные в гл. 7 программы для машины SADSAC всегда располагались в памяти, начиная с нулевой ячейки. Поскольку в каждый момент времени только один пользователь работал на машине, то это было разумным ограничением. Однако подавляющее большинство ЭВМ работает в мультипрограммном режиме, и поэтому две различные программы не могут обе начинаться с нулевой ячейки.

Необходимо также, чтобы библиотечные подпрограммы были доступны всем пользователям. Если значительное число отдельных подпрограмм одновременно загружается в память, мы не можем каждой из них всегда отводить одну и ту же область памяти. Если двум подпрограммам были бы отведены перекрывающиеся области, то они взаимно исключали бы друг друга и главная программа могла бы работать лишь с одной из них.

Решением этих проблем явилось создание *перемещаемого* кода программ, т. е. такого кода, который можно располагать в любых свободных блоках оперативной памяти.

Рассмотрим, например, команду

```
RD2 READ  LIST
```

приведенную во второй SADSAC-программе в гл. 7 (пятая команда программы). Если каждая команда занимает одно слово памяти, то машинный код данной команды попадет в пятое слово. Предположим, что программа была перемещена в машине с большей памятью. Если она теперь начинается в слове 1001, то команда READ будет расположена в слове 1005; если программа начинается в слове 7491, команда READ окажется в слове 7495. Отсюда следует, что в команде

```
STOR  RD2
```

должна быть ссылка на слово 1005 или 7495 соответственно для двух рассмотренных случаев (или на другие слова в зависимости от места расположения программы). Начальный адрес программы называют ее *точкой загрузки*. Процесс корректировки ссылок, связанный с изменением точки загрузки программы, называется *перемещением* программы.

Большинство ЭВМ располагает аппаратными средствами, которые помогают перемещать программы. Часто для этого используется внутренняя ячейка памяти, называемая «регистром перемещений». Компилятор создает объектный код, в котором все ссылки внутри программы вычисляются таким образом, как если бы программа начиналась с ячейки ноль. Когда программа располагается в памяти, адрес ее точки загрузки заносится в регистр перемещений. Каждый раз, когда во время выполнения программы встречается внутренняя ссылка, содержимое регистра перемещений прибавляется к адресу, вычисленному компилятором.

Внешние ссылки — ссылки к точкам, расположенным вне программы или подпрограммы,— обрабатываются иначе. Напомним, что каждая подпрограмма компилируется отдельно от других подпрограмм. Рассмотрим пример обращения из подпрограммы SUB1 к другой подпрограмме SUB2. При компиляции подпрограммы SUB1 невозможно предугадать, где будет расположен код подпрограммы SUB2 во время выполнения программы: он может непосредственно предшествовать коду подпрограммы SUB1, непосредственно следовать за ним или оказаться в какой-либо удаленной области памяти. Компилятор должен пометить те команды внутри подпрограммы SUB1, в которых содержатся ссылки на подпрограмму SUB2, чтобы их можно было скорректировать для получения истинного адреса, когда обе подпрограммы будут загружены в память.

Подпрограммы SUB1 и SUB2 можно перемещать несколькими способами. Во-первых, при каждом обращении к SUB2 в регистр перемещений можно заносить адрес SUB2. При передаче управления из SUB2 на SUB1 в регистре перемещений должен быть восстановлен адрес SUB1. Второй способ состоит в выполнении частичного перемещения, при котором главная программа и ее подпрограммы объединяются для создания так называемого «загрузочного модуля». При частичном перемещении необходимо вычислить смещение SUB2 от начала загрузочного модуля и прибавить это смещение к любым ссылкам внутри подпрограммы SUB2. Во время выполнения программы регистр перемещений содержит адрес, с которого начинается загрузочный модуль, и этот адрес прибавляется к любым ссылкам внутри загрузочного модуля. Такой способ позволяет раздельно перемещать подпрограммы до их выполнения.

Описанные здесь два процесса представляют собой загрузку и редактирование связей. *Загрузка* — это процесс, в ходе которого объектная программа располагается в памяти и перемещается в ней. *Редактирование связей* — это процесс организации связей объектной программы с теми программными модулями, которые она вызывает (или которые ее вызывают).

8.5. Оверлейные структуры

В предыдущем разделе упоминалось, что подпрограммы представляют собой раздельные перемещаемые модули. Поскольку подпрограммы компилируются раздельно, изменение в одной подпрограмме или главной программе требует повторной компиляции только одного, изменяемого модуля, а не целиком всей программы. Другим важным достоинством раздельного перемещения является то, что подпрограммы можно загрузить в любые свободные области памяти. Посмотрим, как проявляются эти полезные свойства при работе с большими программами.

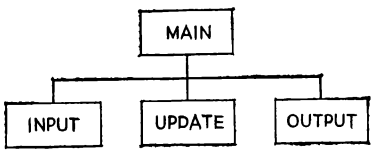


Рис. 8.1. Трехфазная программа.

Рассмотрим программу, основные действия которой могут быть разделены на три фазы: (1) инициализация рабочих областей и чтение набора начальных данных; (2) чтение и обработка информации, изменяющей начальные данные; (3) реорганизация и вывод скорректированных данных для последующих сеансов обработки. Каждая фаза независима от других. Программа может быть построена в виде «драйвера» (главной программы) и трех подпрограмм, которые вводят, корректируют и записывают измененные данные. Таким образом, программу можно представить в виде дерева, изображенного на рис. 8.1, где MAIN, INPUT, UPDATE и OUTPUT — имена соответственно главной программы, подпрограммы ввода, коррекции и вывода данных.

Если задачи сложные, для хранения всей программы может потребоваться большая область памяти. На самом деле размер такой программы может превышать размер имеющейся в наличии памяти. Даже в том случае, если эта программа уместилась бы в оперативной памяти, желательно минимизировать размер требуемой для нее области, так как во многих операционных системах приоритет заданий обратно пропорционален потребности в памяти.

Заметим, что, как только подпрограмма INPUT отработала, она больше не нужна в оперативной памяти. Фактически подпрограммы UPDATE и INPUT, так же как подпрограммы UPDATE и OUTPUT, независимы друг от друга и могут использовать ту же самую область памяти. *Оверлейная структура* (структура с перекрытием) позволяет размещать такие модули в одной и той же области. Что для этого необходимо? Необходимо иметь:

вспомогательную память, такую, как магнитный диск, для хранения копий всех трех модулей; оперативную память, в которой могла бы уместиться наибольшая из трех подпрограмм; подпрограмму супервизора для пересылок модулей по запросам в и из оперативной памяти. Будем говорить, что программа, построенная таким образом, имеет оверлейную структуру, поскольку один модуль занимает ту же область, которая ранее была занята другими модулями.

Предположим, что подпрограммы имеют следующие размеры:

MAIN 30K

INPUT 30K UPDATE 48K OUTPUT 32K

(K — сокращенная запись числа 1024, обычно используется как мера памяти ЭВМ. Приближенное значение 1K=1000 служит хорошей оценкой при определении размера программы.) Тогда для этой программы вместо требуемых 140K слов (30+30+48+32)K оперативной памяти при простой структуре с перекрытием достаточно иметь память размером только 78K слов (30K для

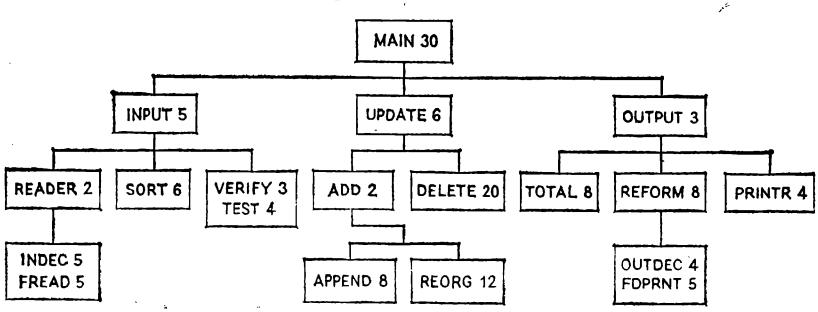


Рис. 8.2. Структура сложной программы.

главной программы плюс 38K для наибольшей из трех подпрограмм). Мы говорим о такого типа оверлейной структуре как о «древовидной структуре». Главная программа является корнем дерева, а все подпрограммы, которые непосредственно вызываются из главной программы,— это ее прямые преемники. Существует единственный путь от корня к любому другому узлу в дереве. Каждый раз, когда подпрограмма находится в памяти, все подпрограммы, расположенные на пути от корня к этой подпрограмме, также должны находиться в памяти.

Оверлейная структура может быть более сложной, чем на приведенном выше рисунке. Например, она может иметь форму, представленную на рис. 8.2 (размеры модулей показаны рядом с их именами).

На этой схеме подпрограмма READER и ее преемники INDEC и FREAD независимы от подпрограммы SORT, которая

в свою очередь независима от подпрограмм VERIFY и TEST. Объем памяти, резервируемой для этих трех секций, равен наибольшей из них (подпрограмма READER и ее преемники). Аналогично для двух преемников подпрограммы UPDATE можно отвести одну область памяти. Заметим, что часть этой области делится между подпрограммами APPEND и REORG. Таким образом, мы видим, что оверлейная область может сама содержать одну или несколько оверлейных подобластей. Вычислим объем памяти, требуемый для всей оверлейной структуры, снизу вверх. Прежде всего определим размер каждого концевого узла дерева (т. е. узла, не имеющего преемников). Затем к размеру каждого предшественника концевого узла добавим размер его наибольшего преемника. Продолжим этот процесс, пока не будет вычислен размер памяти для всей структуры. Размеры всех модулей показаны на рисунке. Область памяти, требуемая для READER, составляет 12К; сюда входит размер узла-преемника плюс размер самой подпрограммы READER. Объем памяти, необходимый для UPDATE, равен 26К, поскольку ее размер равен 6К, а размеры областей двух ее преемников составляют 14К и 20К соответственно. Продолжая таким образом вычисления, получим общий объем памяти, требуемый для программы с такой структурой. Он равен 56К.

Оверлейные структуры влекут за собой некоторые дополнительные расходы. Во-первых, необходимо дополнительное время для приведения объектной программы к оверлейной форме перед ее выполнением. Может понадобиться дополнительная внешняя память во время выполнения программы, поскольку должна быть сохранена полная копия программы на каких-либо устройствах прямого доступа. Так как в ходе выполнения в память могут быть вызваны многие модули структуры, потребуется некоторое время для их загрузки. Наконец, подпрограмма супервизора, управляющая вызовами в оверлейной структуре, требует память как для себя самой, так и для своих таблиц.

Из этих источников дополнительных расходов время, которое уходит на создание оверлейной структуры, обычно не имеет большого значения, поскольку это одноразовая операция. Требуемая область вспомогательной памяти также не вызывает принципиальных затруднений, поскольку емкость вспомогательной памяти в большинстве ЭВМ обычно намного больше, чем оперативной. И наконец, дополнительные затраты,связанные с управлением вызовами, обычно также не имеют значения, поскольку таблицы относительно малы и в ряде ЭВМ оверлейный супервизор не занимает место в памяти пользователя, а написан таким образом, что с одной копией могут работать все пользователи.

Существенным, однако, оказывается дополнительное время, требуемое для обработки вызовов между оверлейными подпро-

граммами. Программист должен спланировать оверлейную структуру так, чтобы минимизировать это время. Если подпрограмма SUB1 будет часто обращаться к подпрограммам SUB2A и SUB2B, то следует расположить SUB2A и SUB2B вдоль того же пути, так чтобы обращение к ним могло выполняться без излишних обменов в памяти. Другой способ планирования оверлейной структуры состоит в помещении подпрограммы, используемой несколькими другими подпрограммами, ближе к корню оверлейного дерева, и тогда такая подпрограмма в момент вызова будет уже в памяти.

Отметим, что в зависимости от обстоятельств некоторые сегменты в выполняемой программе могут или не могут быть оверлейными. Предположим, что какой-то сегмент содержит подпрограмму со следующими инструкциями:

```
DATA      I/1/
          .
          .
          .
          .
        I = 5
```

В ряде «диалектов» Фортрана, если переменная I принимает значение 5, то она сохраняет его в последующих обращениях. Однако в оверлейной структуре, если этот сегмент многократно замещается в памяти и затем вновь вызывается, новая копия, вызванная с диска, может иметь первоначальное значение I, равное 1. Используя оверлейные структуры, программисты должны внимательно следить за тем, чтобы любые значения, которые надо сохранить, попадали в общие блоки. Общие блоки располагают в областях памяти, которые могут быть перекрыты только в том случае, когда отпадает надобность во всех данных, содержащихся в этих блоках.

8.6. Организация оверлейной структуры в DEC-10

LINC-10 — это программа, которая создает оверлейные структуры для ЭВМ DEC-10. Для программы LINC-10 оверлей должен быть построен в форме дерева, как это было описано выше. Если выполняется подпрограмма, находящаяся в некотором узле дерева, то все подпрограммы, расположенные на пути от корня до этого узла, также должны присутствовать в памяти. Корневой узел всегда должен находиться в памяти.

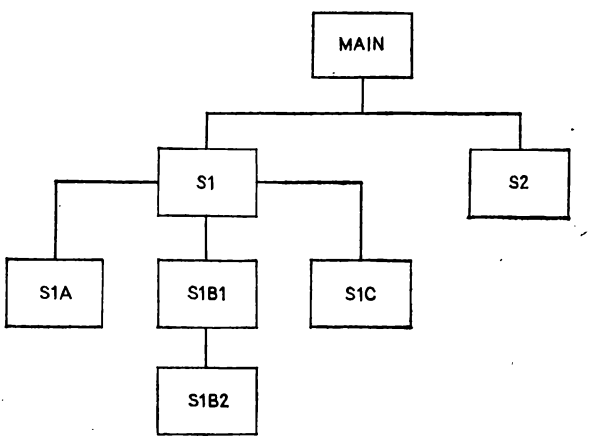
Для организации оверлейной структуры используются следующие ключевые слова:

/OVERLAY — указывает, что должна быть создана оверлейная структура.

/LINK — указывает на окончание определенного узла оверлейного дерева. За этим ключевым словом может также следовать *:имя* — любое имя, посредством которого программист мог бы позднее сослаться на этот узел (чтобы задать его преемников).

/NODE — указывает редактору на определенный узел дерева, для того чтобы определить других преемников этого узла. За этим ключевым словом может последовать один из трех индикаторов позиции, а именно :0, если необходимо вернуться к корню дерева; *:—n*, где *n* — целое число, если необходимо вернуться к *n*-му предшественнику данного узла; или *:имя*, где *имя* — какое-либо имя, если необходимо вернуться к узлу *имя* для подготовки его преемников.

Пользователь перечисляет имена подпрограмм, которые должны находиться в каждом узле, и затем выдает команду /LINK. При интерпретации этой команды строится определенный узел оверлейной структуры и начинается обработка первого преемника данного узла. Например, дерево вида



может быть построено с помощью следующей последовательности команд:

```
/OVERLAY.MAIN/LINK
    S1/LINK:ONE
    S1A/LINK
    /NODE:-1 S1B1/LINK
     S1B2/LINK
    /NODE:ONE S1C/LINK
    /NODE:0 S2/LINK/GO
```

8.7. Организация оверлейной структуры в системах IBM 360-370

Оверлейные структуры в системе DEC-10 имеют тот недостаток, что подпрограмма, которая нужна двум различным узлам, должна быть расположена ближе к корню дерева, чем оба этих узла, или каждый из этих узлов должен содержать копию подпрограммы. Если имеется много таких независимых подпрограмм, то может получиться очень сложное или чрезмерно громоздкое дерево.

В системе IBM эта проблема решена с помощью мультирегионального оверлея. Каждый регион представляет собой эквивалент отдельной оверлейной структуры. В структуре, представленной на рис. 8.3, MAIN и SUB1 или MAIN и SUB2 могут находиться в памяти одновременно с любой из подпрограмм SUBA, SUBB или SUBC.

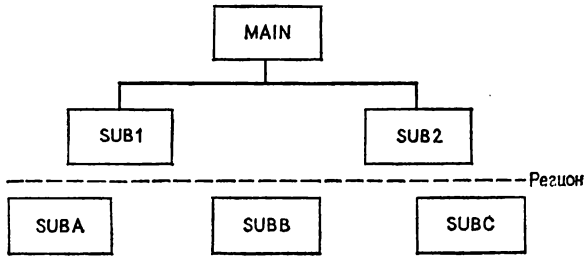


Рис. 8.3. Многорегиональная оверлейная структура.

Узлы SUBA, SUBB и SUBC могут иметь преемников точно так же, как если бы они были преемниками SUB1 или SUB2. Но, поскольку SUBA, SUBB и SUBC образуют отдельный регион, они фактически становятся преемниками обоих узлов SUB1 и SUB2.

Редактор связей — это программа в системе IBM для создания оверлейной структуры. При обращении к редактору связей используются следующие инструкции:

INSERT — объединить один или несколько объектных модулей как часть оверлейного узла. За этой инструкцией располагаются имена одной или нескольких подпрограмм, которые должны быть частью текущего узла.

OVERLAY — указать редактору связей на определенный уровень в оверлейной структуре. Эта инструкция называет узел; когда это имя используется в последующих инструкциях OVERLAY, то эти инструкции определяют другие узлы на том же уровне, на котором первый узел был первоначально идентифицирован по имени. Инструкция

OVERLAY может также определять узел в качестве региона. В этом случае инструкция OVERLAY имеет форму OVERLAY *имя* (REGION).

NAME — дать имя всему загрузочному модулю как программе.

Предположим вы хотите создать оверлейную структуру, соответствующую дереву, приведенному на рис. 8.4:

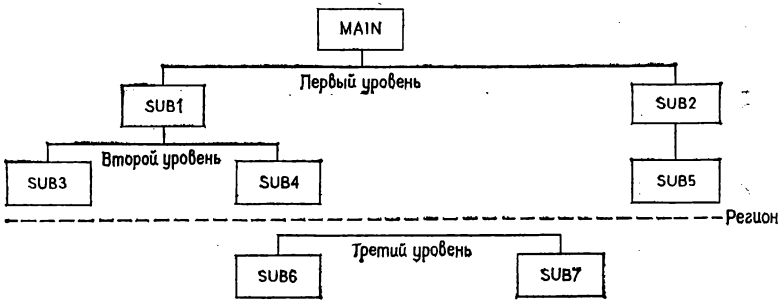


Рис. 8.4. Оверлейное дерево.

Эта оверлейная структура может быть построена с помощью следующих инструкций:

```
INSERT MAIN
OVERLAY ONE
INSERT SUB1
OVERLAY TWO
INSERT SUB3
OVERLAY TWO
INSERT SUB4
OVERLAY ONE
INSERT SUB2,SUB5
OVERLAY THREE(REGION)
INSERT SUB6
OVERLAY THREE
INSERT SUB7
NAME OVERSTR
```

Эта последовательность инструкций позволяет создать оверлейную структуру, соответствующую приведенной на рис. 8.4.

Приложение 1

ВСТРОЕННЫЕ ПОДПРОГРАММЫ-ФУНКЦИИ ОБЩЕГО НАЗНАЧЕНИЯ

Функция	Имя	Определение	Аргументы: Число	Аргументы: Тип	Тип результата
Максимум/минимум от аргументов	MAX0	$y=\max(x_1, \ldots, x_n)$	$\geqslant 2$	INTEGER	INTEGER
	MAX1		$\geqslant 2$	REAL	INTEGER
	AMAX0		$\geqslant 2$	INTEGER	REAL
	AMAX1		$\geqslant 2$	REAL	REAL
	MIN0	$y=\min(x_1, \ldots, x_n)$	$\geqslant 2$	INTEGER	INTEGER
	MIN1		$\geqslant 2$	REAL	INTEGER
	AMIN0		$\geqslant 2$	INTEGER	REAL
	AMIN1		$\geqslant 2$	REAL	REAL
Усечение	AINT	$y=(\operatorname{sign} x)\cdot n$	1	REAL	REAL
	INT	где n наибольшее целое $\leqslant \lvert x \rvert$	1	REAL	INTEGER
Значение по модулю	MOD	$y=$остаток	2	INTEGER	INTEGER
	AMOD	$x_1 \div x_2$ (т. е. $y=x_1 \bmod x_2$)	2	REAL	REAL
Плавающая точка	FLOAT	$y=$вещественное, эквивалентное x	1	INTEGER	REAL
Фиксированная точка	IFIX	$y=$целое, эквивалентное x	1	REAL	INTEGER
Знак числа	ISIGN	$y=(\operatorname{sign} x_2)\cdot x_1$	2	INTEGER	INTEGER
	SIGN		2	REAL	REAL
Положительная разность	IDIM	$y=x_1-\min(x_1, x_2)$	2	INTEGER	INTEGER
	DIM	(т. е. $y=x_1-x_2$, если $(x_1-x_2)>0$, иначе $=0$)	2	REAL	REAL

Функция	Имя	Определение	Аргументы: Число	Аргументы: Тип	Тип результата
Логарифм	ALOG	$y=\ln x$	1	REAL	REAL
	ALOG 10	$y=\log_{10} x$	1	REAL	REAL
Экспонента	EXP	$y=e^x$	1	REAL	REAL
Корень квадратный	SQRT	$y=\sqrt{x}$	1	REAL	REAL
Синус	SIN	$y=\sin x$	1	REAL	REAL
Косинус	COS	$y=\cos x$	1	REAL	REAL
Арктангенс	ATAN	$y=\operatorname{arctg} x$	1	REAL	REAL
Абсолютное	IABS	$y=\lvert x\rvert$	1	INTEGER	INTEGER
Значение	ABS		1	REAL	REAL

Приложение 2

РАЗЛИЧИЯ МЕЖДУ ФОРТРАНОМ 4 И ФОРТРАНОМ 77

В приложении приведены изменения в языке Фортран, введенные в версии Фортран 77. Приложение следует рассматривать не как полное описание языка Фортран 77, а лишь как перечень изменений по отношению к материалу, изложенному в гл. 0. Мы не вводим здесь новых инструкций, а только показываем, как изменяется синтаксис или смысл инструкций, представленных в гл. 0. Правда, с одним исключением. Фортран 77 содержит конструкцию IF-THEN-ELSE; поскольку в ряде компиляторов Фортрана эта конструкция допустима в той или иной форме как расширение Фортрана 4, мы приводим описание новой инструкции IF.

Материал приложения распределен в том же порядке, что и в гл. 0 этой книги. Номера разделов в приложении совпадают с номерами соответствующих разделов гл. 0. Так что вы можете легко найти сноску в приложении каждый раз, когда в гл. 0 встречается абзац, выделенный жирными точками.

0.2.1. Формат инструкции

В Фортране 77 допустимы пустые строки. Они обрабатываются как комментарии (т. е. игнорируются). Комментарии могут начинаться или с буквы «С», или с символа «*», звездочка.

0.2.2. Типы данных

Литерные константы в форме Н (*n*Н *цепочка*) разрешается использовать только в инструкциях FORMAT. Литерные константы, заключенные в апострофы (*'цепочка'*), являются теперь общепринятой формой. Однако многие компиляторы в целях совместимости, по-видимому, будут допускать литерные константы типа Н в инструкциях CALL и DATA.

Индекс может быть любым разрешенным арифметическим выражением; если выражение не относится к типу целых, то значение будет усечено для использования в качестве индекса.

0.2.3. Операции

Включены две новые логические операции .EQV. и .NEQV. , которые устанавливают соответственно отношение «эквивалентно» или «не эквивалентно». Если L1 и L2 — два логических выражения, то (L1 .EQV. L2) будет истинно, если и L1,и L2 или истинны, или ложны, т. е. .EQV. истинно только тогда, когда оба его аргумента имеют одно и то же значение. Выражение (L1 .NEQV. L2) означает то же самое, что и выражение (.NOT. (L1 .EQV. L2)), т. е. (L1 .NEQV. L2)) будет истинно только тогда, когда L1 и L2 имеют различные значения.

Для работы с литерами добавлена новая операция //. Операция // означет конкатенацию, или объединение двух цепочек. Если S1 имеет значение 'DOG' и S2 — значение 'HOUSE', то S1//S2 будет иметь значение 'DOGHOUSE'. Аналогично можно расчленить цепочку на части посредством операции ссылки на подцепочку. Для указания подцепочки в скобки заключаются два выражения, указывающие позиции первой и последней из тех литер, которые должны быть включены в подцепочку. Например, если S3 имеет значение 'DOGHOUSE', то S3 (1, 3) выделяет 'DOG', а S3(4,5)—'HO'. Если ссылка на массив и ссылка на подцепочку относятся к одной и той же переменной, то индексы массива стоят первыми, т. е. A(I, J) (K, L) — это подцепочка от K-й до L-й литеры элемента в массиве A(I, J).

0.3.1. Декларации

Разрешены переменные типа CHARACTER. Синтаксис декларации CHARACTER тот же самый, что описан в разд. 3.8. Тип CHARACTER может быть также указан в инструкции IMPLICIT или FUNCTION. Во всех трех случаях декларация CHARACTER может иметь вид CHARACTER*n, где n — целая константа, определяющая длину цепочки литер.

Размер массива может быть задан в виде $A(d_1 : d_2)$, где d_1 и d_2— соответственно нижний и верхний пределы значений индексов. Например, DIMENSION A(3 : 10) объявляет восьмиэлементный массив, содержащий элементы A(3), A(4), . . ., A(10).

0.3.1. Инструкции DATA

Значение не обязательно должно быть того же типа, что и инициализируемая *переменная*; преобразование будет происходить точно так, как если бы *переменная* и *значение* участвовали в инструкции присваивания. Можно также инициализировать любую часть массива, указав $A(i_1) : A(i_2)$ в *списке переменных.*

Такая запись означает, что значения должны быть присвоены всем элементам массива A, начиная с элемента с индексом i_1 и кончая элементом с индексом i_2.

0.5.2. Бесформатный ввод («ввод, управляемый списком»)

Разрешена форма инструкции ввода

READ (*устр-ввода*, *) *список*.

0.5.3. Бесформатный вывод («вывод, управляемый списком»)

Разрешена форма инструкции вывода

WRITE (*устр-вывода*, *) *список*.

0.5.3, 0.5.6. Бесформатный вывод, форматный WRITE

Список переменных в инструкции WRITE может также включать константы и выражения.

0.6.2. Вычисляемый GOTO

Индекс вычисляемого GOTO может быть любым целым выражением. Если значение выражения больше количества содержащихся в списке меток инструкций или если это значение не положительно, выполнение программы продолжается с инструкции, которая следует за вычисляемым GOTO.

0.6.3. Инструкция IF

Добавлены четыре новые инструкции:

IF (*выраж*) THEN END IF
ELSE IF (*выраж*) THEN ELSE

Каждая конструкция IF-THEN должна быть связана с соответствующей инструкцией END IF. Между IF-THEN и соответствующей ей инструкцией END IF может быть любое количество инструкций ELSE IF-THEN и не более одной инструкции ELSE. Указанные группы должны быть правильно вложены, т. е. если одна группа IF содержится внутри другой группы IF, то все инструкции первой (вложенной) группы IF должны находиться между IF-THEN и END IF второй (внешней) группы IF. Переходы в группу IF извне не допустимы.

Смысл этих конструкций аналогичен описанным в разд. 1.3.2 конструкциям IF-THEN-ELSE.

0.6.4. Инструкция цикла DO

Параметры *начальн*, *конечн* и *шаг* могут теперь быть любым целым выражением. «Счетчик итерации» устанавливается перед началом выполнения цикла DO. Его начальное значение — MAXO (((*конечн* — *начальн* + *шаг*)/*шаг*), 0). Значение счетчика итераций — это число повторений цикла DO; заметим, что счетчик может иметь нулевое значение. В этом случае никаких инструкций в цикле не выполняется. Шаг цикла может быть отрицательным. Можно изменить значения параметров цикла во время его выполнения. Однако это не повлияет на количество повторений цикла, так как счетчик итераций вычисляется прежде, чем тело цикла выполнится в первый раз. Нельзя передавать управление извне в тело цикла DO.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Адрес (address) 16
Альтернативные выходы из подпрограммы (alternate exits from subprogram) 123
Аргумент функции [*см.* Параметр]
Арифметическое выражение (arithmetic expression) 18
Арифметическое и логическое устройство (arithmetic and logical unit) 287, 289
Ассоциированная переменная (associated variable) 210

Байт (byte) 279
Бесформатный ввод/вывод (free format input/output) 28, 159, 194, 318
Бит (bit) 13, 265
— четности (parity bit) 185
Блок комментариев — «шапка» (header comments) 69
— COMMON 135
— — несогласующиеся определения (inconsistent definitions) 140
— — помеченный 141
Блокирование записей (blocking) 186
Буферизация вывода [*см.* Спулинг]

Ввод/вывод (input/output) 27, 157, 184
Вещественная константа (real constant) 13
Возврат из подпрограммы (return) 45
Возвращение результата подпрограммы 52
— — функции 50
Восьмеричная система счисления (octal numbering system) 276
Временна́я сложность алгоритма (time complexity) 93
Встроенные функции (built-in functions) 48, 314
Вызов подпрограммы (call) 45, 52
— функции 51
Выражение (expression) 18
— вычисление (evaluation) 20
Вычисляемый переход (computed GOTO) 78

Генератор случайных чисел (random number generator) 146
— мультипликативный (multiplicative) 147
Глобальные данные (global data) 135
Голова списка (head of a list) 227

Дамп (dump) 81
Данные (data) 265
Двоичное представление (binary representation)
— — вещественных чисел (of real numbers) 269, 278
— — — — с двойной точностью (double precision) 275, 278
— — целых чисел (of integer numbers)
— — — — в дополнительном коде (twos complement) 267, 277, 283
— — — — в обратном коде (ones complement) 267
— — — — в прямом коде (sign and magnitude) 266
Двойное слово (double word) 279
Декларация (declaration) 21
— типа переменной (variable type) 22
— — — неявная (on default) 23
Деление на ноль (division by zero) 271
Дерево (tree) 245
— бинарное поиска 246
— сбалансированное (balanced) 246
Диалект языка (language dialect) 12
Документация внешняя (external documentation) 66
— внутренняя (internal) 68
Дополнительные входы в подпрограмму (multiple entries) 125
Дуга (arc) 245

Загрузка программы (program loading) 306
Защита от ошибок (defensive programming) 80

Имя массива (array's name) 15
— переменной (variable) 15
Индекс массива (subscript) 15
Инструкции (statements)
— безусловного перехода (unconditional branch) 38
— ввода/вывода (input/output) 210, 318
— вычисляемого перехода (computed branch) 39, 318
— для работы с дисками 210
— — — магнитными лентами 193
— локального управления последовательностью действий (local control of execution) 37

— неисполняемые (nonexecutable) 21
— присваивания (assignment) 26
— условного перехода (conditional branch) 39
— цикла (iteration, automatic looping) 41—42

Кодирование литер (character encodings) 156
Коллизия при хешировании (collision) 130, 151
Колонка продолжения (continuation field) 13
Комментарий (comments) 13, 70
— заголовок (header) 69
Компилятор (compiler) 12, 304
— FORTRAN-10 12
— WATFIV 12, 168
Корень дерева (root of a tree) 246

Лентопротяжное устройство (magnetic tape unit) 191
Лист дерева (leaf of a tree) 246
Литера (character) 155
Литерал (literal) 32
Литерная константа (character constant) 14, 158
Логическая запись (logical record) 186
— константа (logical constant) 14
Логическое выражение (logical expression) 19
Локальные данные (local data) 116, 134

Магнитные диски (magnetic disks) 206
— ленты (tapes) 185
Маркер конца записи (end of record marker) 188
— — файла (end of file) 191
Массив (array) 15
Метка инструкции (label) 12
— файла 191
Методы поиска (search methods) 93, 129
— сортировки (sorting) 95
— — быстрой (quick sorting) 97
— — «пузырьком» (bubble) 96
Моделирование (simulation) 251
— автозаправочной станции 252—255
— ЭВМ SADSAC 295
Монитор (monitor) 302
Монте-Карло метод 147
Мультипрограммирование (multiprogramming) 302

Неисполняемая инструкция (nonexecutable statement) 21
Неявный цикл (implied DO) 28
Номер инструкции [*см.* Метка инструкции]
Нумерация последовательности перфокарт (sequence number) 13

Общий блок [*см.* Блок COMMON]
Объектный код (object code) 304
Оверлей (overlay) 307, 311
Оглавление тома (VTOC) 208
Операция (operations)
— арифметические (arithmetic operators) 17
— замещения (replacement) 167
— конкатенации (concatenation) 165, 317
— логические 17
— над литерами (character operations) 159, 165
— над стеком (stack operations) 229
— отношения (relation operators) 17
— присваивания (assignment) 26
— с очередью (queue operations) 233
Операционная система (operating system) 198, 301
Определение подпрограммы (subprogram definition) 52, 118
— функции 49
Отладка программы (debugging) 81, 86
Отладочные инструкции 83—85
Очередь (queue) 232
— связанная (linked) 241

Пакетная обработка (batch) 302
Память оперативная (main memory) 184, 287
Параметр подпрограммы (argument) 45, 115
— — фактический (actual) 46, 52, 118
— — формальный (dummy) 46, 53, 115
Передача параметров (parameter transmission) 52, 118
— — по адресу 120
— — — значению (by value) 121
— — — имени (by name) 119
Перемещаемый код (relocatable code) 305
Переполнение (overflow) 271
— порядка (exponent overflow) 271
— стека 231
Перфокарта (card) 12
Поддерево левое/правое узла (subtree) 246
Подпрограмма (subprogram) 44, 115
Подпрограмма-функция 45—47

Полемика с GOTO 75
Полуслово (half word) 279
Портативность программы (portability) 76
Последовательный доступ (sequential access) 185, 209
— список 228
Потеря порядка (exponent underflow) 272
Правила приоритета операций (precedence) 18—19
Предпроцессор 76
Представление данных (data representation) 265
— последовательное (sequential) 235
— связанное (linked) 237
Предшественник — узел (predecessor) 246
Преемник — узел (successor) 246
Программа, управляемая таблицей (table-driven) 102
Программный модуль (program module) 45
— счетчик (program counter) 289
Произвольный доступ (random access) 210
Простая переменная (simple variable) 15
Процедура (procedure) 115
Прямой доступ (direct access) 185, 206
Псевдослучайная последовательность чисел 146

Разделение времени (time sharing) 303
Размерность массива (dimension) 21
— — регулируемая (adjustable) 55, 129
Разряд защиты (guard digit) 278
Расширение языка программирования (language extension) 76
Регистр адреса 288
— данных 288
— перемещений (relocation register) 306
Редактирование связей (linking) 306
Руководство для пользователей (user's guide) 66
— по логике работы программы (Program Logic Manual — PLM) 67, 171

Связанное представление списка (linked list representation) 238
Символ управления (control character) 36
Системный каталог (system catalog) 213
Слово памяти (word) 13, 276
Случайные числа (random numbers) 146
Согласование параметров (agreement of arguments) 46—47
Список (list) 227
— параметров подпрограммы 46
— свободного пространства (available space) 240

— связанный (linked list) 238
— со многими связями (multilinked) 243
Ссылка внешняя (external reference) 306
Стандарт Фортрана ANSI-66 11
Статическое распределение памяти (static allocation) 16
Стек (stack) 228
— связанный 240
Стиль программирования 64
Структура вычислительной машины (computer structure) 287
— данных 227
— — динамическая (dynamic) 231
— программы 74
— управления последовательностью действий (control structure) 76
— — — — DO-WHILE 78
— — — — IF-THEN-ELSE 77
— — — — REPEAT-UNTIL 78

Таблица символов (symbol table) 16, 248
Тип данных (data type) 13
— переменной (variable type) 15, 281
Точка загрузки (load point) 305
Трассировка (trace flow) 81

Узел (node) 245
Указатель связанного списка (link) 238
— очереди (queue pointer) 234
— стека (stack pointer) 230
Уровень дерева (level of a tree) 246
Устройства ввода/вывода (input/output units, devices) 34, 206, 287
Учебная программа ACM-68 (ACM curriculum 68) 7

Файл данных (data file) 191
Физическая запись (physical record) 186
Форматный ввод/вывод (formatted input/output) 31
Формула доступа (storage mapping function) 16
Фортран-77 7, 11, 168, 316
Фортрана инструкции:
BACKSPACE 196
CHARACTER 168, 317
COMMON 135
CONTINUE 42
DATA 24, 134, 317

DEFINE FILE 210
DIMENSION 21
DO 41, 77
END 25
ENDFILE 196
END IF 318
ENTRY 125
EQUIVALENCE 103
EXTERNAL 126
FIND 218
FORMAT 31, 32
FUNCTION 49
GO TO 38, 74
IF 39
IF-THEN 318
IMPLICIT 23
INTEGER 22
LOGICAL 22
PRINT 30
READ 29, 33, 211
REAL 22
RETURN 49
REWIND 196
STOP 38
SUBROUTINE 52
WRITE 35, 211
Функция 116
Функция-оператор (statement-function) 114

Хеширование (hash coding) 129
— коллизия (collision) 130

Целая константа (integer constant) 13
Центральный процессор (central processor) 287, 289
Цепочка битов (bit string) 156
— литер (character string) 14, 155
Цикл (iteration, automatic looping) 77

Шестнадцатеричная система счисления (hexadecimal numbering system) 276

ЭВМ DEC-10, 7, 24, 208—214, 283, 311
— IBM/360-370 9, 188, 208—214, 277
— SADSAC 290
Элемент данных (data element) 13
— массива 15
Эффективность программы (efficiency) 90

Язык ассемблера (assembly language) 292
— машинный 291
— управления заданиями (job control language) 198, 213

ОГЛАВЛЕНИЕ

Предисловие редактора перевода 5

Предисловие 7

Глава 0. Предварительные сведения 11

0.1. Обзор 11
0.2. Основные понятия и определения 12
0.2.1. Запись инструкций Фортрана 12
0.2.2. Типы данных 13
0.2.3. Операции 17
0.2.4. Выражения 18
0.2.5. Условные обозначения 20
0.3. Неисполняемые инструкции 21
0.3.1. Декларации 21
0.3.2. Инструкция END 25
0.4. Инструкция присваивания 26
0.5. Инструкции ввода/вывода 27
0.5.1. Списки переменных для ввода и вывода 28
0.5.2. Бесформатный ввод 29
0.5.3. Бесформатный вывод 30
0.5.4. Инструкция FORMAT и форматный ввод/вывод 31
0.5.5. Форматный ввод 33
0.5.6. Форматный вывод 35
0.6. Инструкции локального управления последовательностью действий 37
0.6.1. Прекращение выполнения программы 38
0.6.2. Безусловный переход 38
0.6.3. Условные переходы 39
0.6.4. Инструкции цикла 41
0.7. Подпрограммы 44
0.7.1. Введение 44
0.7.2. Списки параметров 46
0.7.3. Встроенные функции 48
0.7.4. Подпрограммы-функции, определенные пользователем 49
0.7.5. Подпрограммы типа SUBROUTINE 52
0.8. Пример 55
Упражнения 61

Глава 1. Стиль программирования 64

1.1. Пример 64
1.2. Документация 66
1.2.1. Внешняя документация 66
1.2.2. Внутренняя документация 68
1.2.3. Дополнительные рекомендации 71
1.3. Структура программы 74
1.3.1. Дискуссия об инструкции GO TO 74

1.3.2. Структуры управления 76
1.4. Защита от ошибок при программировании 79
1.5. Методы отладки 80
1.5.1. Средства отладки 81
1.5.2. Как же все-таки производят отладку? 86
1.6. Эффективность 90
1.6.1. Эффективность в смысле машинного времени/памяти . 90
1.6.2. Эффективность алгоритма 92
1.7. Изменяемость программы 100
1.8. Разделяемые ячейки памяти: инструкция EQUIVALENCE . . 103
1.9. Переработанный пример 108
Ссылки на литературу 114

Глава 2. Процедуры 115

2.1. Элементарные характеристики процедур Фортрана 115
2.1.1. Функции 116
2.1.2. Подпрограммы 118
2.2. Передача параметров 118
2.2.1. Основные правила связи формальных и фактических параметров 118
2.2.2. Методика обеспечения связи параметров 119
2.2.3 Передача параметров по имени 119
2.2.4. Передача параметров по значению 121
2.3. Дополнительные сведения о процедурах 122
2.3.1. Альтернативные выходы из подпрограмм 123
2.3.2. Дополнительные входы в процедуру 125
2.3.3. Процедуры в роли параметров 126
2.3.4. Переменные размерности 129
2.4. Поиск в таблицах при использовании хеширования 129
2.5. Инструкции DATA в подпрограммах 134
2.6. COMMON — инструкция Фортрана для объявления данных глобальными 134
2.6.1. Локальные и глобальные данные 134
2.6.2. Стандартное применение инструкции COMMON . . . 135
2.6.3. Несогласующиеся определения COMMON 140
2.6.4. Блок COMMON с именем 141
2.6.5. Взаимосвязь инструкций COMMON и EQUIVALENCE 142
2.6.6. Инициализация элементов данных в общих блоках . . 143
2.7. Функция-оператор 144
2.8. Заключительный пример — генераторы случайных чисел . 146
2.8.1. Понятие псевдослучайной последовательности 146
2.8.2. Мультипликативный генератор случайных чисел. . . . 147
2.8.3. Интегрирование по методу Монте-Карло 147
Упражнения 150

Глава 3. Программирование нечисленных задач 155

3.1. Литеры в качестве данных 155
3.2. Кодирование литер 155
3.3. Ввод и вывод литерных данных 157
3.4. Литерные константы 158
3.5. Операции над литерами 159
3.6. Пример: бесформатный ввод целых 159
3.7. Другие методы работы с литерами 165
3.8. Данные литерного типа в языках WATFIV и Фортран 77 . . 168
3.8.1. Литерные константы 169
3.9. Примеры 171
Упражнения 181

Глава 4. Ввод/вывод: магнитные ленты и диски 184
4.1. Физическая организация магнитных лент 185
4.1.1. Запись данных 185
4.1.2. Записи фиксированной и переменной длины 187
4.1.3. Проблема обновления записей 190
4.1.4. Файлы данных 191
4.1.5. Физические характеристики магнитной ленты 191
4.1.6. Скорость передачи данных на магнитной ленте 192
4.2. Инструкции Фортрана для работы с магнитной лентой . . . 193
4.2.1. Форматный ввод/вывод на магнитной ленте 193
4.2.2. Бесформатный ввод/вывод на магнитной ленте 194
4.2.3. Инструкция BACKSPACE 196
4.2.4. Инструкция ENDFILE 196
4.2.5. Инструкция REWIND 196
4.3. Работа с магнитными лентами: системы IBM и DEC 197
4.3.1. Назначение устройствам в системе IBM OS/JCL . . . 198
4.3.2. Назначение устройствам в системе DEC-10 204
4.4. Устройства прямого доступа 206
4.4.1. Характеристики некоторых дисковых устройств . . . 208
4.4.2. Управление дисковой памятью в OS IBM 360-370 . . . 208
4.4.3. Управление дисковой памятью в системе DEC-10 . . . 209
4.4.4. Последовательная обработка дисковых файлов 209
4.4.5. Обработка дисковых файлов с произвольным доступом 210
4.5. Назначение файлам устройств прямого доступа — примеры 212
4.5.1. Назначение устройствам прямого доступа в системе OS/JCL IBM 212
4.5.2. Назначение устройствам прямого доступа в системе DEC-10 214
4.6. Пример. Система поиска книг по ключевым словам 215
Упражнения 224

Глава 5. Структура данных 227
5.1. Списки 227
5.2. Стеки 228
5.3. Очереди 232
5.4. Последовательное распределение памяти с указателями. . . 235
5.5. Связанное распределение 237
5.6. Списки со многими связями 242
5.6.1. Несколько упорядочений одного списка 242
5.6.2. Списки с двумя связями 244
5.7. Деревья 245
5.7.1. Бинарные деревья 246
5.7.2. Пример использования бинарного дерева поиска — работа с таблицей символов 248
5.8. Пример работы со списками — моделирование автозаправочной станции 251
5.8.1. О моделировании 251
5.8.2. Моделируемая система 251
5.8.3. Планирование и обработка событий 253
Упражнения 263

Глава 6. Представление данных в машине 265
6.1. Двоичное представление целых чисел 265
6.1.1. Прямой код 266
6.1.2. Обратный код 267
6.1.3. Дополнительный код 268
6.2. Двоичное представление вещественных чисел 268

6.3. Ошибки и потеря точности — целые числа 271
6.4. Ошибки и потеря точности — вещественные числа 272
6.5. Двойная точность 274
6.6. Восьмеричные и шестнадцатеричные числа 275
6.7. Представление чисел в IBM 360-370 277
6.8. Организация памяти в машинах IBM 360-370 279
6.8.1. Байты, полуслова, слова и двойные слова 279
6.8.2. Типы переменных 281
6.8.3. Переменные типа LOGICAL*1 и действия с литерами . 282
6.9. Представление чисел в ЭВМ DEC-10 283
6.10. Организация памяти ЭВМ DEC-10 284
Упражнения 285

Глава 7. Структура простой ЭВМ 287
7.1. Устройства ЭВМ 287
7.1.1. Оперативная память 287
7.1.2. Центральный процессор 289
7.1.3. Арифметическое и логическое устройство. 289
7.2. Простейшая машина 290
7.2.1. Подробное описание машины SADSAC 290
7.2.2. Язык ассемблера для машины SADSAC 292
7.2.3. Моделирование машины SADSAC 294
Упражнения 300

Глава 8. Эффективное программирование с использованием средств операционной системы 301
8.1. Операционные системы 301
8.2. Компиляторы 303
8.3. Объектный код 304
8.4. Перемещение программ 305
8.5. Оверлейные структуры 307
8.6. Организация оверлейной структуры в DEC-10 310
8.7. Организация оверлейной структуры в системах IBM 360-370 312

Приложение 1. Встроенные подпрограммы-функции общего назначения 314

Приложение 2. Различия между Фортраном 4 и Фортраном 77 316

Предметный указатель 320

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство «Мир».

Ч. Хьюз, Ч. Пфлигер, Л. Роуз

МЕТОДЫ ПРОГРАММИРОВАНИЯ: КУРС НА ОСНОВЕ ФОРТРАНА

Научный редактор К. Г. Батаев
Мл. научный редактор Л. С. Суркова
Художник В. М. Новоселова
Художественный редактор В. И. Шаповалов
Технический редактор Л. П. Бирюкова
Корректор М. А. Смирнов

ИБ № 2405

Сдано в набор 09.09.80.
Подписано к печати 09.01.81.
Формат 60×90 1/16.
Бумага типографская № 2.
Гарнитура литературная. Печать высокая.
Объем 10.50 бум. л. Усл. печ. л. 21
Уч.-изд. л. 19,66. Изд. № 1/1020.
Тираж 30.000 экз. Зак. № 2043. Цена 1 р. 40 к.

ИЗДАТЕЛЬСТВО «МИР»
Москва, 1-й Рижский пер., 2.
Ордена Октябрьской Революции и ордена Трудового Красного Знамени Первая Образцовая типография имени А. А. Жданова Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. Москва, М-54, Валовая, 28

Издательство „Мир“ предлагает книги по программированию и математическому обеспечению ЭВМ

Ахо У., Ульман Дж. **Теория синтаксического анализа и трансляции.** т. 1. 1978. 2 р. 90 к., т. 2. 1978. 2 р. 40 к.

Бауэр Ф., Гнац Р., Хилл У. **Информатика. Задачи и решения.** 1978. 1 р. 30 к.

Вайнгартен Ф. У. **Трансляция языков программирования.** 1977. 75 к.

Гилман Л., Роуз А. **Курс АПЛ; диалоговый подход.** 1979. 2 р. 10 к.

Гловински Р., Лионс Ж.-Л. **Численное исследование вариационных неравенств.** 1979. 2 р. 50 к.

Гренандер У. **Лекции по теории образов. т. 1. Синтез образов.** 1979. 1 р. 80 к.

Грисуолд Р., Поудж Дж., Полонски И. **Язык программирования Снобол-4.** 1980. 95 к.

Драммонд М. **Методы оценки и измерений дискретных вычислительных систем.** 1977. 1 р. 81 к.

Кнут Д. **Искусство программирования для ЭВМ. т. 2. Получисленные алгоритмы.** 1977. 3 р. 80 к.

Кнут Д. **Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск.** 1978. 4 р. 80 к.

Коддингтон А. **Ускоренный курс Кобола.** 1974. 1 р. 18 к.

Линдси Ч. **Неформальное введение в Алгол-68.** 1973. 1 р. 72 к.

Маджинис Дж. **Программирование на стандартном Коболе.** 1979. 2 р. 30 к.

Маурер У. **Введение в программирование на языке ЛИСП.** 1976. 92 к.

Машины Тьюринга и рекурсивные функции. 1972. 89 к.

Оллонгрен А. **Определение языков программирования интерпретирующими автоматами.** 1977. 1 р. 20 к.

Партхасаратхи Т., Рагхаван Т. **Некоторые вопросы теории игр двух лиц.** 1974. 1 р. 34 к.

Пейган Ф. **Практическое руководство по Алголу-68.** 1979. 85 к.

Пересмотренное сообщение об Алголе-68. 1980. 2 р. 60 к.

Перечислительные задачи комбинаторного анализа. 1979. 2 р.

Пратт Т. **Языки программирования. Разработка и реализация.** 1979. 2 р. 90 к.

Психология машинного зрения. 1978. 1 р. 80 к.

Розенмюллер И. **Кооперативные игры и рынки.** 1974. 64 к.

Сложность вычислений и алгоритмов. 1974. 1 р. 96 к.

Холл П. **Вычислительные структуры. Введение в нечисленное программирование.** 1978. 95 к.

Эти книги Вы можете приобрести в магазинах книготоргов, распространяющих научно-техническую литературу. Если в ближайшем от Вас магазине этих книг не окажется, заказ можно направить по адресу:

121019 Москва, просп. Калинина, 26, п/я 42, Магазин № 200 «Московский Дом книги».

103031 Москва, Петровка, 16, Магазин № 8 «Техническая книга»,

191040 Ленинград, Пушкинская ул., 2, магазин № 5 «Техническая книга».

Книги будут высланы наложенным платежом (без задатка).

В издательстве «Мир» выходит в свет новая книга

Мейер Б., Бодуэн К. **Методы программирования:** Пер. с франц. 1981, 44 л., 3 р. 50 к.

В монографии французских ученых систематически излагаются основные понятия информатики, обсуждаются трудные проблемы методологии программирования, дается сравнение известных языков программирования: Фортрана, Алгола, PL/1 и др. Изложение сопровождается оригинальными упражнениями (с решениями).

Для профессиональных программистов, желающих овладеть современными методами программирования. Может служить учебным пособием по языкам программирования и алгоритмам.

Если Вы желаете приобрести эту книгу, оставьте в книжном магазине предварительный заказ. Своевременное оформление заказа гарантирует Вам приобретение нужной книги.

В издательстве «Мир» выходит в свет новая книга

Турский В. **Методология программирования на ЭВМ:** Пер. с англ., 1981, 16 л., 1 р. 50 к.

Монография написана профессором Варшавского университета и отражает коллективный опыт руководимой им Рабочей группы по методологии программирования (в ее составе известные специалисты: Н. Вирт, У. Дал, Э. Дейкстра, Д. Грис, К. Хоар и др.). Ее предметом являются три крупных раздела: фундаментальные конструкции программирования, модульная структура программ и процесс построения программ. Изложение последовательное, ясное, однако для чтения книги необходимо иметь навыки программирования.

Книга предназначена программистам, желающим углубить понимание предмета, и будет полезна аспирантам и студентам в качестве дополнительного материала к основному курсу программирования.

Если Вы желаете приобрести эту книгу, оставьте в книжном магазине предварительный заказ. Своевременное оформление заказа гарантирует Вам приобретение нужной книги.

1 р. 40 к.

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

ИЗДАТЕЛЬСТВО МИР